SUBJECT: ARTIFICIAL INTELLIGENCE	
COURSE CODE: MCA-23	**AUTHOR:**
LESSON NO. 1	**VETTER:**
ARTIFICIAL INTELLIGENCE AND ITS APPLICATION	

UNIT STRUCTURE

1.0 Objectives

1.1 Introduction

1.2 Applications of AI

- 1.2.1 Games
- 1.2.2 Theorem Proving
- 1.2.3 Natural Language Processing
- 1.2.4 Vision and Speech Processing
- 1.2.5 Robotics
- 1.2.6 Expert Systems

1.3 AI Techniques

- 1.3.1 Knowledge Representation
- 1.3.2 Search Technique

1.4 Search Knowledge

1.5 Abstraction

1.6 Production System

1.7 Summary

1.8 Keywords

1.9 Check Your Progress

1.10 Reference/Suggested Reading

1.0 Objective

The objective of this lesson is to provide an introduction to the definitions, techniques, components and applications of Artificial Intelligence. Upon completion

of this lesson students should able to answer the AI problems, Techniques, and games. This lesson also gives an overview about expert system, search knowledge and abstraction.

1.1 Introduction

Artificial Intelligence (AI) is the area of computer science focusing on creating machines that can engage on behaviors that humans consider intelligent. The ability to create intelligent machines has intrigued humans since ancient times, and today with the advent of the computer and 50 years of research into AI programming techniques, the dream of smart machines is becoming a reality. Researchers are creating systems which can mimic human thought, understand speech, beat the best human chess player, and countless other feats never before possible.

What is Artificial Intelligence (AI)?

According to Elaine Rich, "Artificial Intelligence "

"Artificial Intelligence is the study of how to make computers do things at which, at the moment, people are better".

In what way computer & Human Being are better?

Computers	**Human Being**
1. Numerical Computation is fast	1. Numerical Computation is slow
2. Large Information Storage Area	2. Small Information Storage Area
3. Fast Repetitive Operations	3. Slow Repetitive Operations
4. Numeric Processing	5. Symbolic Processing
5.Computers are just Machine (Performed Mechanical "Mindless" Activities)	4. Human Being is intelligent (make sense from environment)

Other Definitions of Artificial Intelligence

According to Avron Barr and Edward A. Feigenbaum, "The Handbook of Artificial Intelligence", the goal of AI is to develop intelligent computers. Here intelligent computers mean that emulates intelligent behavior in humans.

"Artificial Intelligence is the part of computer science with designing intelligent computer systems, that is, systems that exhibit the characteristics we associate with intelligence in human behavior."

Other definitions of AI are mainly concerned with symbolic processing, heuristics, and pattern matching.

Symbolic Processing

According to Bruce Buchanan and Edward Shortliffe" Rule Based Expert Systems" (reading MA: Addison-Wesley, 1984), p.3.

"Artificial Intelligence is that branch of computer science dealing with symbolic, non algorithmic methods of problem solving."

Heuristics

According to Bruce Buchanan and Encyclopedic Britannica, heuristics as a key element of a Artificial Intelligence:

"Artificial Intelligence is branch of computer science that deals with ways of representing knowledge using symbols rather than numbers and with rules-of-thumb or heuristics, methods for processing information."

A heuristics is the "rule of thumb" that helps us to determine how to proceed.

Pattern Matching

According to Brattle Research Corporation, Artificial Intelligence and Fifth Generation Computer Technologies, focuses on definition of Artificial Intelligence relating to pattern matching.

"In simplified terms, Artificial Intelligence works with the pattern matching methods which attempts to describe objects, events, or processes in terms of their qualitative features and logical and computational relationships."

Here this definition focuses on the use of pattern matching techniques in an attempt to discover the relationships between activities just as human do.

1.2 Application of Artificial Intelligence

1.2.1.0 Games

Game playing is a search problem Defined by

– Initial state

– Successor function

– Goal test

– Path cost / utility / payoff function

Games provide a structured task wherein success or failure can be measured with latest effort. Game playing shares the property that people who do them well are considered to be displaying intelligence. There are two major components of game playing, viz., a plausible move generator, and a static evaluation function generator. Plausible move generator is used to expand or generates only selected moves. Static evaluation function generator, based on heuristics generates the static evaluation function value for each & every move that is being made.

1.2.1.1 Chess

AI-based game playing programs combine intelligence with entertainment. On game with strong AI ties is chess. World-champion chess playing programs can see ahead twenty plus moves in advance for each move they make. In addition, the programs have an ability to get progressably better over time because of the ability to learn. Chess programs do not play chess as humans do. In three minutes, Deep Thought (a master program) considers 126 million moves, while human chessmaster on average considers less than 2 moves. Herbert Simon suggested that human chess masters are familiar with favorable board positions, and the relationship with thousands of pieces in small areas. Computers on the other hand, do not take hunches into account. The next move comes from exhaustive searches into all moves, and the consequences of the moves based on prior learning. Chess programs, running on Cray super computers have attained a rating of 2600 (senior master), in the range of Gary Kasparov, the Russian world champion.

1.2.1.2 Characteristics of game playing

- "Unpredictable" opponent.
    - Solution is a strategy specifying a move for every possible opponent reply.
- Time limits.
    - Unlikely to find goal, must approximate.

1.2.2 Theorem Proving

Theorem proving has the property that people who do them well are considered to be displaying intelligence. The Logic Theorist was an early attempt to prove mathematical theorems. It was able to prove several theorems from the Qussells

Principia Mathematica. Gelernters' theorem prover explored another area of mathematics: geometry. There are three types of problems in A.I. Ignorable problems, in which solution steps can be ignored; recoverable problems in which solution steps can be undone; irrecoverable in which solution steps cannot be undone. Theorem proving falls into the first category i.e. it is ignorable suppose we are trying to solve a theorem; we proceed by first proving a lemma that we think will be useful. Eventually we realize that the lemma is not help at all. In this case we can simply ignore that lemma, and can start from beginning.

There are two basics methods of theory proving.

- Start with the given axioms, use the rules of inference and prove the theorem.
- Prove that the negation of the result cannot be TRUE.

1.2.3 Natural Language Processing

The utility of computers is often limited by communication difficulties. The effective use of a computer traditionally has involved the use of a programming language or a set of commands that you must use to communicate with the computer. The goal of natural language processing is to enable people and computer to communicate in a "natural "(human) language, such as a English, rather than in a computer language.

The field of natural language processing is divided into the two sub-fields of:

- Natural language understanding, which investigates methods of allowing computer to comprehend instruction given in ordinary English so that computers can understand people more easily.
- Natural language generation, which strives to have computers produce ordinary English language so that people can understand computers more easily.

1.2.4 Vision and Speech Processing

The focus of natural language processing is to enable computers to communicate interactively with English words and sentences that are typed on paper or displayed on a screen. However, the primary interactive method of communication used by humans is not reading and writing; it is speech.

The goal of speech processing research is to allow computers to understand human speech so that they can hear our voices and recognize the words we are speaking. Speech recognition research seeks to advance the goal of natural language processing by simplifying the process of interactive communication between people and

computers. It is a simple task to attach a camera to computer so that the computer can receive visual images. It has proven to be a far more difficult task, however, to interpret those images so that the computer can understand exactly what it is seeing. People generally use vision as their primary means of sensing their environment; we generally see more than we hear, feel, smell or taste. The goal of computer vision research is to give computers this same powerful facility for understanding their surroundings. Currently, one of the primary uses of computer vision is in the area of robotics.

1.2.5 Robotics

A robot is an electro-mechanical device that can be programmed to perform manual tasks. The Robotic Industries Association formally defines a robot as "a reprogrammable multi-functional manipulator designed to move material, parts, tools or specialized devices through variable programmed motions for the performance of a variety of tasks." An "intelligent" robot includes some kind of sensory apparatus, such as a camera, that allows it to respond to changes in its environment, rather than just to follow instructions "mindlessly."

1.2.6 Expert System

An expert system is a computer program designed to act as an expert in a particular domain (area of expertise). Also known as a knowledge-based system, an expert system typically includes a sizable knowledge base, consisting of facts about the domain and heuristics (rules) for applying those facts. Expert system currently is designed to assist experts, not to replace them. They have proven to be useful in diverse areas such as computer system configuration.

A ``knowledge engineer" interviews experts in a certain domain and tries to embody their knowledge in a computer program for carrying out some task. How well this works depends on whether the intellectual mechanisms required for the task are within the present state of AI. When this turned out not to be so, there were many disappointing results. One of the first expert systems was MYCIN in 1974, which diagnosed bacterial infections of the blood and suggested treatments. It did better than medical students or practicing doctors, provided its limitations were observed. Namely, its ontology included bacteria, symptoms, and treatments and did not include patients, doctors, hospitals, death, recovery, and events occurring in time. Its

interactions depended on a single patient being considered. Since the experts consulted by the knowledge engineers knew about patients, doctors, death, recovery, etc., it is clear that the knowledge engineers forced what the experts told them into a predetermined framework. In the present state of AI, this has to be true. The usefulness of current expert systems depends on their users having common sense.

1.3 AI Techniques

There are various techniques that have evolved that can be applied to a variety of AI tasks - these will be the focus of this course. These techniques are concerned with how we represent, manipulate and reason with knowledge in order to solve problems.

1.3.1 Knowledge Representation

Knowledge representation is crucial. One of the clearest results of artificial intelligence research so far is that solving even apparently simple problems requires lots of knowledge. Really understanding a single sentence requires extensive knowledge both of language and of the context. For example, today's (4th Nov) headline ``It's President Clinton'' can only be interpreted reasonably if you know it's the day after the American elections. [Yes, these notes are a bit out of date]. Really understanding a visual scene similarly requires knowledge of the kinds of objects in the scene. Solving problems in a particular domain generally requires knowledge of the objects in the domain and knowledge of how to reason in that domain - both these types of knowledge must be represented. Knowledge must be represented efficiently, and in a meaningful way. Efficiency is important, as it would be impossible (or at least impractical) to explicitly represent every fact that you might ever need. There are just so many potentially useful facts, most of which you would never even think of. You have to be able to infer new facts from your existing knowledge, as and when needed, and capture general abstractions, which represent general features of sets of objects in the world.

Knowledge must be meaningfully represented so that we know how it relates back to the real world. A knowledge representation scheme provides a mapping from features of the world to a formal language. (The formal language will just capture certain aspects of the world, which we believe are important to our problem - we may of course miss out crucial aspects and so fail to really solve our problem, like ignoring friction in a mechanics problem). Anyway, when we manipulate that formal language

using a computer we want to make sure that we still have meaningful expressions, which can be mapped back to the real world. This is what we mean when we talk about the semantics of representation languages.

1.3.2 Search

Another crucial general technique required when writing AI programs is *search*. Often there is no direct way to find a solution to some problem. However, you do know how to generate possibilities. For example, in solving a puzzle you might know all the possible moves, but not the sequence that would lead to a solution. When working out how to get somewhere you might know all the roads/buses/trains, just not the best route to get you to your destination quickly. Developing good ways to search through these possibilities for a good solution is therefore vital. *Brute force* techniques, where you generate and try out every possible solution may work, but are often very inefficient, as there are just too many possibilities to try. *Heuristic* techniques are often better, where you only try the options, which you think (based on your current best guess) are most likely to lead to a good solution.

1.4 Search Knowledge

In order to solve the complex problems encountered in artificial intelligence, one needs both a large amount of knowledge and some mechanisms for manipulating that knowledge to create solutions to new problems. That is if we have knowledge that it is sufficient to solve a problem, we have to search our goal in that knowledge. To search a knowledge base efficiently, it is necessary to represent the knowledge base in a systematic way so that it can be searched easily. Knowledge searching is a basic problem in Artificial Intelligence. The knowledge can be represented either in the form of facts or in some formalism. A major concept is that while intelligent programs recognize search, search is computationally intractable unless it is constrained by knowledge about the world. In large knowledge bases that contain thousands of rules, the intractability of search is an overriding concern. When there are many possible paths of reasoning, it is clear that fruitless ones not be pursued. Knowledge about path most likely to lead quickly to a goal state is often called search control knowledge.

1.5 Abstraction

Abstraction a mental facility that permits humans to view real-world problems with varying degrees of details depending on the current context of the problem. Abstraction means to hide the details of something. For example, if we want to compute the square root of a number then we simply call the function sort in C. We do not need to know the implementation details of this function. Early attempts to do this involved the use of macro-operators, in which large operators we built from smaller one's. But in this approach, no details were eliminated from actual description of the operators. A better approach was developed in the ABSTRIPS system, which actually planned in a hierarchy of abstraction spaces, in each of which preconditions at a lower level of abstraction, was ignored.

1.6 Production Systems

A production system is a system that adapts a system with production rules.

A production system consists of:

- A set of rules, each consisting of a left side and a right hand side. Left hand side or pattern determines the applicability of the rule and a right side describes the operation to be performed if the rule is applied.
- One or more knowledge/databases that contain whatever information is appropriate for the particular task. Some parts of the database may be permanent, while other parts of it may pertain only to the solution of the current problem. The information in these databases may be structured in any appropriate way.
- A control strategy that specifies the order in which the rules will be compared to the database and a way of resolving the conflicts that arise when several rules match at once.
- A rule applier.

Production System also encompasses a family of general production system interpreters, including:

- Basic production system languages, such as OPS5 The OPS (said to be short for "Official Production System") family was developed in the late 1970s by Charles Forgy while at Carnegie Mellon University.

- More complex, often hybrid systems called *expert system shells,* which provide complete (relatively speaking) environments for the construction of knowledge-based expert systems.
- General problem-solving architectures like SOAR [Laird *et al.*, 1987], SOAR (Security Orchestration, Automation, and Response) refers to a collection of software solutions and tools that allow organizations to streamline security operations in three key areas: threat and vulnerability management, incident response, and security operations automation. a system based on a specific set of cognitively motivated hypotheses about the nature of problem solving.

Above systems provide the overall architecture of a production system and allow the programmer to write rules that define particular problems to be solved.

In order to solve a problem, firstly we must reduce it to one for which a precise statement can be given. This is done by defining the problem's state space, which includes the start and goal states and a set of operators for moving around in that space. The problem can then be solved by searching for a path through the space from an initial state to a goal state. The process of solving the problem can usefully be modelled as a production system. In production system we have to choose the appropriate control structure so that the search can be as efficient as possible.

The important roles played by production systems include a powerful knowledge representation scheme. A production system not only represents knowledge but also action. It acts as a bridge between AI and expert systems. Production system provides a language in which the representation of expert knowledge is very natural. We can represent knowledge in a production system as a set of rules of the form If (condition) THEN (condition) along with a control system and a database. The control system serves as a rule interpreter and sequencer. The database acts as a context buffer, which records the conditions evaluated by the rules and information on which the rules act. The production rules are also known as condition – action, antecedent – consequent, pattern – action, situation – response, feedback – result pairs.

For example,

If (you have an exam tomorrow)

THEN (study the whole night)

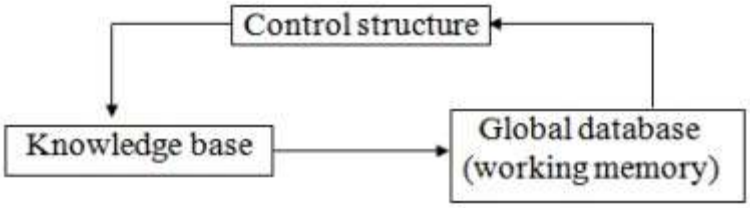


Figure 2.6 Architecture of Production System

FEATURES OF PRODUCTION SYSTEM

Some of the main features of production system are:

1. **Expressiveness and intuitiveness**: In real world, many times situation comes like "if this happen-you will do that", "if this is so-then this should happen" and many more. The production rules essentially tell us what to do in a given situation.
2. **Simplicity:** The structure of each sentence in a production system is unique and uniform as they use "IF-THEN" structure. This structure provides simplicity in knowledge representation. This feature of production system improves the readability of production rules.
3. **Modularity:** This means production rule code the knowledge available in discrete pieces. Information can be treated as a collection of independent facts which may be added or deleted from the system with essentially no deleterious side effects.
4. **Modifiability:** This means the facility of modifying rules. It allows the development of production rules in a skeletal form first and then it is accurate to suit a specific application.
5. **Knowledge intensive:** The knowledge base of production system stores pure knowledge. This part does not contain any type of control or programming information. Each production rule is normally written as an English sentence; the problem of semantics is solved by the very structure of the representation.

DISADVANTAGES OF PRODUCTION SYSTEM

1. **Opacity:** This problem is generated by the combination of production rules. The opacity is generated because of less prioritization of rules. More priority to a rule has the less opacity.
2. **Inefficiency:** During execution of a program several rules may active. A well devised control strategy reduces this problem. As the rules of the production system are large in number and they are hardly written in hierarchical manner, it

requires some forms of complex search through all the production rules for each cycle of control program.

3. **Absence of learning:** Rule based production systems do not store the result of the problem for future use. Hence, it does not exhibit any type of learning capabilities. So for each time for a particular problem, some new solutions may come.
4. **Conflict resolution:** The rules in a production system should not have any type of conflict operations. When a new rule is added to a database, it should ensure that it does not have any conflicts with the existing rules.

Classifications of Production Systems

Production system describes the operations that can be performed in a search for a solution to the problem. These are four classifications.

- A monotonic production system
- A non-monotonic production system
- A partially commutative production system
- A commutative production system.

Monotonic Production System: It's a production system in which the application of a rule never prevents the later application of another rule, that could have also been applied at the time the first rule was selected.

Partially Commutative Production System: It's a type of production system in which the application of a sequence of rules transforms state X into state Y, then any permutation of those rules that is allowable also transforms state x into state Y. Theorem proving falls under the monotonic partially communicative system.

Blocks world and 8 puzzle problems like chemical analysis and synthesis come under monotonic, not partially commutative systems.

Although, playing the game of bridge comes under non-monotonic, not partially commutative system. For any problem, several production systems do exist. Some will be efficient than others.

Non-Monotonic Production Systems: These are useful for solving ignorable problems. These systems are important for man implementation standpoint because they can be implemented without the ability to backtrack to previous states when it is discovered that an incorrect path was followed. This production system increases the

efficiency since it is not necessary to keep track of the changes made in the search process.

Commutative Systems: They are usually useful for problems in which changes occur but can be reversed and in which the order of operation is not critical for example the, 8 puzzle problem. Production systems that are not usually not partially commutative are useful for many problems in which irreversible changes occur, such as chemical analysis. When dealing with such systems, the order in which operations are performed is very important and hence correct decisions must be made at the first time itself.

1.7 Summary

In this chapter, we have defined AI, other definitions of AI & terms closely related to the field. Artificial Intelligence (AI) is the part of computer science concerned with designing intelligent computer systems, that is, systems that exhibit the characteristics. We associate with intelligence in human behavior, other definition of AI are concerned with symbolic processing, heuristics, and pattern matching. Artificial intelligence problems appear to have very little in common except that they are hard. Areas of AI research have been evolving continually. However, as more people identify research-taking place in a particular area as AI, that are will tend to remain a part of AI. This could result in a more static definition of Artificial Intelligence. Currently, the most well known area of AI research is expert system, where programs include expert level knowledge of a particular field in order to assist experts in that field. Artificial Intelligence is best understood as an evolution rather than a revolution, some of popular application areas of AI include games, theorem proving, natural language processing, vision, speech processing, and robotics.

1.8 Keywords

- **Artificial Intelligence (AI):** Artificial Intelligence is branch of computer science that deals with ways of representing knowledge using symbols rather than numbers and with rules-of-thumb or heuristics, methods for processing information

- **Natural Language Processing:** The goal of natural language processing is to enable people and computer to communicate in a “natural “(human) language, such as a English, rather than in a computer language
- **Speech Recognition:** The goal of speech processing research is to allow computers to understand human speech so that they can hear our voices and recognize the words we are speaking. Speech recognition research seeks to advance the goal of natural language processing by simplifying the process of interactive communication between people and computers.
- **Production System:** Production system or production rule system is a computer program typically used to provide some form of artificial intelligence, which consists primarily of a set of rules about behaviour but it also includes the mechanism necessary to follow those rules as the system responds to states of the world.

1.9 Check Your Progress

Q1. A key element of AI is a/an _____, which is a “rule of thumb”.

a. Heuristics
b. Cognition
c. Algorithm
d. Digiton

Q2. One definition of AI focuses on problem solving methods that process:

a. Numbers
b. Symbols
c. Actions
d. Algorithms

Q3 Intelligent planning programs may be of speed value to managers with _____ Responsibilities.

a. Programming
b. Customer source
c. Personal administration
d. Decision making

Q4. What is AI? Explain different definition of AI with different application of AI.

Q5. Write short note on the following: -

a. Robotics
b. Expert system
c. Natural Language Processing
d. Vision of Speech Processing

1.10 Reference/Suggested Reading

- Foundations of Artificial Intelligence and Expert System - V S Janakiraman, K Sarukesi, & P Gopalakrishanan, Macmillan Series.
- Artificial Intelligence – E. Rich and K. Knight
- Principles of Artificial Intelligence – Nilsson
- Expert Systems-Paul Harmon and David King, Wiley Press.
- Rule Based Expert System-Bruce G. Buchanan and Edward H. Shortliffe, eds., Addison Wesley.
- Introduction to Artificial Intelligence and Expert System- Dan W. Patterson, PHI, Feb., 2003.

SUBJECT: ARTIFICIAL INTELLIGENCE	
COURSE CODE: MCA-23	**AUTHOR:**
LESSON NO. 2	**VETTER:**
PROBLEM SOLVING IN AI	

UNIT STRUCTURE

2.0 Objectives

2.1 Introduction

2.2 Defining state space of the problem

2.3 Search Space Control

2.3.1 Depth First Search

2.3.2 Breadth First Search

2.3.3 Heuristic Search Techniques

2.3.4 Hill Climbing

2.3.5 Best First Search

2.3.6 Branch and Bound

2.3.7 A* Algorithm

2.4 Problem Reduction

2.4.1 AND/OR Graph

2.5 Constraints Satisfaction

2.6 Means End Analysis

2.7 minimax Search procedure

2.8 Summary

2.9 Keywords

2.10 Check Your Progress

2.11 Reference/Suggested Reading

2.0 Objective

The objective of this lesson is to provide an overview of problem representation techniques, production system, search space control and hill climbing. This lesson also gives in depth knowledge about the searching techniques. After completion of this lesson, students are able to tackle the problems related to problem representation, production system and searching techniques.

2.1 Introduction

Before a solution can be found, the prime condition is that the problem must be very precisely defined. By defining it properly, one can convert it into the real workable states that are really understood. These states are operated upon by a set of operators and the decision of which operator to be applied, when and where is dictated by the overall control strategy.

Problem must be analysed. Important features land up having an immense impact on the appropriateness of various possible techniques for solving the problem.

Out of the available solutions choose the best problem-solving technique(s) and apply the same to the particular problem.

2.2 Defining state space of the problem

A set of all possible states for a given problem is known as state space of the problem. Representation of states is highly beneficial in AI because they provide all possible states, operations and the goals. If the entire sets of possible states are given, it is possible to trace the path from the initial state to the goal state and identify the sequence of operators necessary for doing it. For example, Problem statement **"Play chess."**

To discuss state space problem, let us take an example of "play chess". Inspite of the fact that there are a many people to whom we could say that and reasonably expect that they will do as we intended, as our request now stands its quite an incomplete

statement of the problem we want solved. To build a program that could "Play chess," first of all we have to specify the initial position of the chessboard, any and every rule that defines the legal move, and the board positions that represent a win for either of the sides. We must also make explicit the previously implicit goal of not only playing a legal game of chess but also goal towards winning the game.

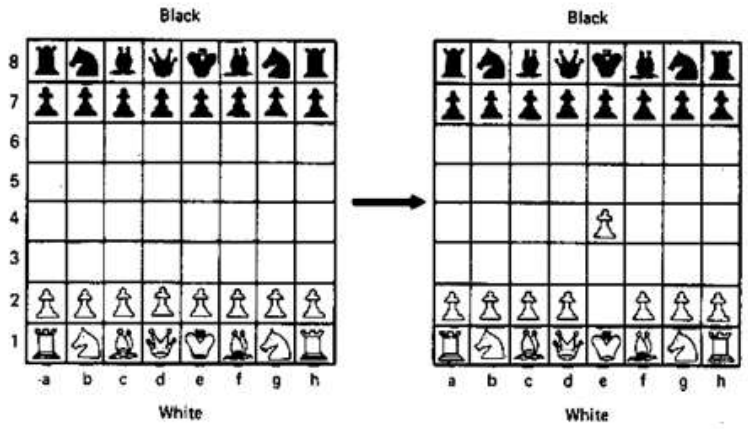


Figure 2.1: One Legal Chess Move

Its quite easy to provide an acceptable complete problem description for the problem "Play chess," The initial position can be described as an 8-by-8 array where each position contains a symbol standing for the appropriate piece in the official chess opening position. Our goal can be defined as any board position in which either the opponent does not have a legal move or opponent's king is under attack. The path for getting the goal state from an initial state is provided by the legal moves. Legal moves are described easily as a set of rules consisting of two parts: a left side that serves as a pattern to be matched against the current board position and a right side that describes the change to be made to the board position to reflect the move. There are several ways in which these rules can be written. For example, we could write a rule such as that shown in Figure 2.1.

In case we write rules like the one above, we have to write a very large number of them since there has to be a separate rule for each of the roughly 10^{120} possible board positions. Using so many rules poses two serious practical difficulties:

- We will not be able to get a complete set of rules. If at all we manage then it is likely to take too long and will certainly be consisting of mistakes.
- Any program will not be able to handle these many rules. Although a hashing scheme could be used to find the relevant rules for each move fairly quickly, just storing that many rules poses serious difficulties.

One way to reduce such problems could possibly be that write the rules describing the legal moves in as general a way as possible. To achieve this we may introduce some convenient notation for describing patterns and substitutions. For example, the rule described in Figure 2.1, as well as many like it, could be written as shown in Figure 2.2. In general, the more efficiently we can describe the rules we need, the less work we will have to do to provide them and the more efficient the program that uses them can be.

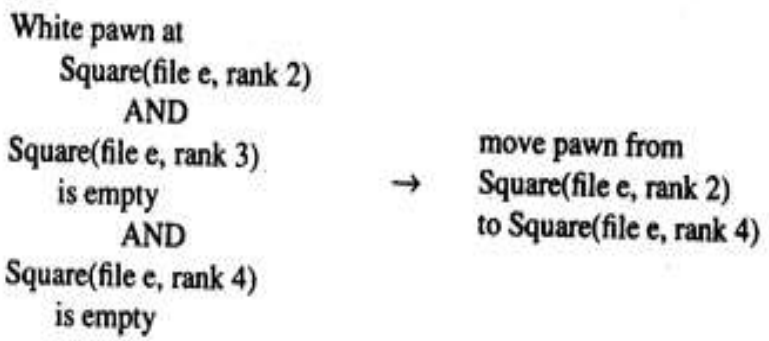


Figure 2.2: Another Way to Describe Chess Moves

Problem of playing chess has just been described as a problem of moving around, in a *state space,* where a legal position represents a state of the board. Then we play chess by starting at an initial state, making use of rules to move from one state to another, and making an effort to end up in one of a set of final states. This state space representation seems natural for chess because the set of states, which corresponds to the set of board positions, is artificial and well organized. This same kind of representation is also useful for naturally occurring, less well-structured problems, although we may need to use more complex structures than a matrix to describe an individual state. The basis of most of the AI methods we discuss here is formed by the

State Space representations. Its structure corresponds to the structure of problem solving in two important ways:

- Representation allows for a formal definition of a problem using a set of permissible operations as the need to convert some given situation into some desired situation.
- We are free to define the process of solving a particular problem as a combination of known techniques, each of which are represented as a rule defining a single step in the space, and search, the general technique of exploring the space to try to find some path from the current state to a goal state.

Search is one of the important processes the solution of hard problems for which none of the direct techniques is available.

2.3 Search Space Control

The next step is to decide which rule to apply next during the process of searching for a solution to a problem. This decision is critical since often more than one rule (and sometimes fewer than one rule) will have its left side match the current state. We can clearly see what a crucial impact they will make on how quickly, and even whether, a problem is finally solved. There are mainly two requirements to of a good control strategy. These are:

1. A good control strategy must cause motion
2. A good control strategy must be systematic: A control strategy is not systematic; we may explore a particular useless sequence of operators several times before we finally find a solution. The requirement that a control strategy be systematic corresponds to the need for global motion (over the course of several steps) as well as for local motion (over the course of a single step). One systematic control strategy for the water jug problem is the following. Construct a tree with the initial state as its root. Generate all the offspring of the root by applying each of the applicable rules to the initial state.

Now, for each leaf node, generate all its successors by applying all the rules that are appropriate. Continuing this process until some rule produces a goal state. This

process, called *breadth-first search*, can be described precisely in the breadth first search algorithm.

2.3 Depth First Search

The searching process in AI can be broadly classified into two major types. Viz. Brute Force Search and Heuristics Search. Brute Force Search do not have any domain specific knowledge. All they need is initial state, the final state and a set of legal operators. Depth-First Search is one the important technique of Brute Force Search.

In Depth-First Search, search begins by expanding the initial node, i.e., by using an operator, generate all successors of the initial node and test them. Let us discuss the working of DFS with the help of the algorithm given below.

Algorithm for Depth-First Search

1. Put the initial node on the list of START.
2. If (START is empty) or (START = GOAL) terminate search.
3. Remove the first node from the list of START. Call this node d.
4. If (d = GOAL) terminate search with success.
5. Else if node d has successors, generate all of them and add them at the beginning of START.
6. Go to step 2.

In DFS the time complexity and space complexity are two important factors that must be considered. As the algorithm and Fig. 2.3 shows, a goal would be reached early if it is on the left hand side of the tree.

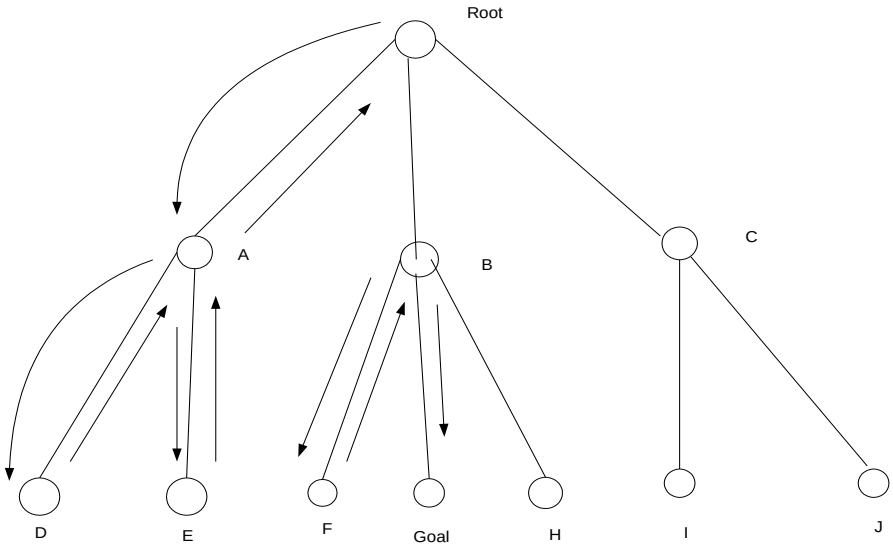


Figure: 2.3 Search tree for Depth-first search

The major drawback of Depth-First Search is the determination of the depth (cut-off depth) until which the search has to proceed. The value of cut-off depth is essential because otherwise the search will go on and on.

2.4 Breadth First Search

Breadth first search is also like depth first search. Here searching progresses level by level. Unlike depth first search, which goes deep into the tree. An operator employed to generate all possible children of a node. Breadth first search being the brute force search generates all the nodes for identifying the goal.

Algorithm for Breadth-First Search

1. Put the initial node on the list of START.
2. If (START is empty) or (START = GOAL) terminate search.
3. Remove the first node from the list of START. Call this node d.
4. If (d = GOAL) terminate search with success.
5. Else if node d has successors, generate all of them and add them at the tail of START.
6. Go to step 2.

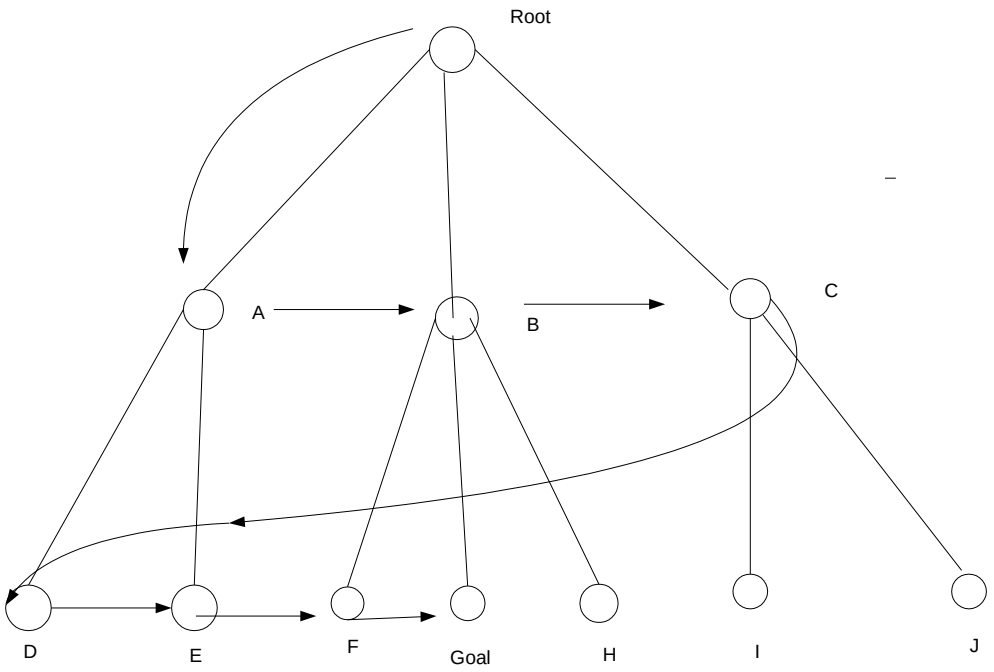


Figure: 2.4 Search tree for Breadth-first search

Similar to brute force search two important factors time-complexity and space-complexity have to be considered here also.

The major problems of this search procedure are:

1. Amount of time needed to generate all the nodes is considerable because of the time complexity.
2. Memory constraint is also a major hurdle because of space complexity.
3. The Searching process remembers all unwanted nodes, which is of no practical use for the search.

2.5Heuristic Search Techniques

The idea of a "heuristic" is a technique, which sometimes will work, but not always. It is sort of like a rule of thumb. Most of what we do in our daily lives involves heuristic solutions to problems. Heuristics are the approximations used to minimize the searching process.

The basic idea of heuristic search is that, rather than trying all possible search paths, you try and focus on paths that seem to be getting you nearer your goal state. Of course, you generally can't be sure that you are really near your goal state – it could

be that you'll have to take some amazingly complicated and circuitous sequence of steps to get there. But we might be able to have a good guess. Heuristics are used to help us make that guess.

To use heuristic search you need an evaluation function (Heuristic function) that scores a node in the search tree according to how close to the target/goal state it seems to be. This will just be a guess, but it should still be useful. For example, for finding a route between two towns a possible evaluation function might be a ``as the crow flies'' distance between the town being considered and the target town. It may turn out that this does not accurately reflect the actual (by road) distance – maybe there aren't any good roads from this town to your target town. However, it provides a quick way of guessing that helps in the search.

Basically heuristic function guides the search process in the most profitable direction by suggesting which path to follow first when more than one is available. The more accurately the heuristic function estimates the true merits of each node in the search tree (or graph), the more direct the solution process. In the extreme, the heuristic function would be so good that essentially no search would be required. The system would move directly to a solution. But for many problems, the cost of computing the value of such a function would outweigh the effort saved in the search process. After all, it would be possible to compute a perfect heuristic function by doing a complete search from the node in question and determining whether it leads to a good solution. Usually there is a trade-off between the cost of evaluating a heuristic function and the savings in search time that the function provides.

There the following algorithms make use of heuristic evaluation function.

- Hill Climbing
- Best First Search
- Constraints Satisfaction

2.6 Hill Climbing

Hill climbing uses a simple heuristic function viz., the amount of distance the node is from the goal. This algorithm is also called Discrete Optimization Algorithm. Let

us discuss the steps involved in the process of Hill Climbing with the help of an algorithm.

Algorithm for Hill Climbing Search

1. Put the initial node on the list of START.
2. If (START is empty) or (STRAT = GOAL) terminate search.
3. Remove the first node from the list of START. Call this node d.
4. If (d = GOAL) terminate search with success.
5. Else if node d has successors, generate all of them. Find out how far they are from the goal node. Sort them by the remaining distance from the goal and add them to the beginning of START.
6. Go to step 2.

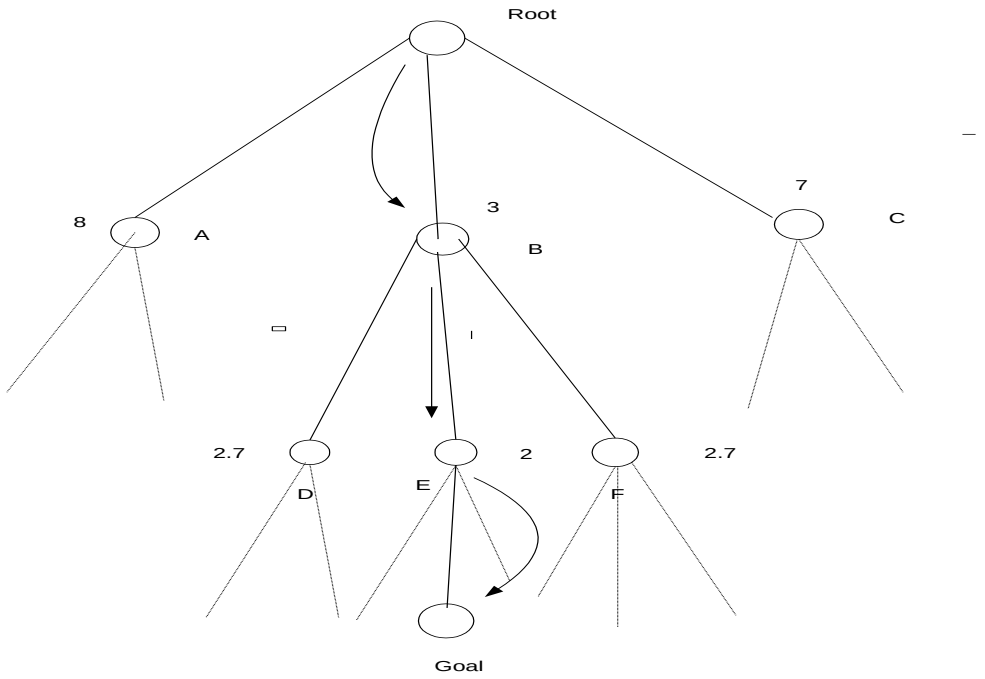


Figure 2.5: Search tree for hill-climbing procedure

Problems of Hill Climbing Technique

Local Maximum: A state that is better than all its neighbours but no so when compared to the states that are farther away.

Plateau: A flat area of search space, in which all the neighbours have the same value.

Ridge: Described as a long and narrow stretch of elevated ground or narrow elevation or raised part running along or across a surface by the Oxford English Dictionary.

Solution to the problems of Hill Climbing Technique

- *Backtracking for local maximum:* Backtracking helps in undoing what has been done so far and permits to try a totally different path to attain the global peak.
- A big jump is the solution to escape from the plateau.
- Trying different paths at the same time is the solution for circumventing ridges.

2.7 Best First Search

Best first search is a little like hill climbing, in that it uses an evaluation function and always chooses the next node to be that with the best score. The heuristic function used here (evaluation function) is an indicator of how far the node is from the goal node. Goal nodes have an evaluation function value of zero.

Algorithm for Best First Search

1. Put the initial node on the list of START.
2. If (START is empty) or (STRAT = GOAL) terminate search.
3. Remove the first node from the list of START. Call this node d.
4. If (d = GOAL) terminate search with success.
5. Else if node d has successors, generate all of them. Find out how far they are from the goal node. Sort all the children generated so far by the remaining distance from the goal.
6. Name this list as START 1.
7. Replace START with START 1.
8. Go to step 2.

The path found by best first search are likely to give solutions faster because it expands a node that seems closer to the goal.

2.8 Branch and Bound

Branch and Bound search technique applies to a problem having a graph search space where more than one alternate path may exist between two nodes. An algorithm for the branch and bound search technique uses a data structure to hold partial paths developed during the search are as follows.

Place the start node of zero path length on the queue.

1. Until the queue is empty or a goal node has been found: (a) determine if the first path in the queue contains a goal node, (b) if the first path contains a goal node exit with success, (c) if the first path does not contain a goal node, remove the path from the queue and form new paths by extending the removed path by one step, (d) compute the cost of the new paths and add them to the queue, I sort the paths on the queue with lowest-cost paths in front.
2. Otherwise, exit with failure.

2.9A* Algorithm

A * algorithm is a searching algorithm that searches for the shortest path between the initial and the final state. It is used in various applications, such as maps. A* is based on using heuristic methods to achieve *optimality* and *completeness*, and is a variant of the best-first algorithm.

When a search algorithm has the property of optimality, it means it is guaranteed to find the best possible solution, in our case the shortest path to the finish state. When a search algorithm has the property of completeness, it means that if a solution to a given problem exists, the algorithm is guaranteed to find it.

Each time A* enters a state, it calculates the cost, $f(n)$ (n being the neighboring node), to travel to all of the neighboring nodes, and then enters the node with the lowest value of $f(n)$.

These values are calculated with the following formula:

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

$g(n)$ being the value of the shortest path from the start node to node *n*, and $h(n)$ being a heuristic approximation of the node's value.

The efficiency of A* is highly dependent on the heuristic value $h(n)$, and depending on the type of problem, we may need to use a different heuristic function for it to find the optimal solution.

Construction of such functions is no easy task and is one of the fundamental problems of AI. The two fundamental properties a heuristic function can have are admissibility and consistency.

Admissibility and Consistency

A given heuristic function $h(n)$ is *admissible* if it never overestimates the real distance between n and the goal node.

Therefore, for every node *n* the following formula applies:

$$h(n) \leq h*(n)$$

$h*(n)$ being the real distance between *n* and the goal node. However, if the function does overestimate the real distance, but never by more than d, we can safely say that the solution that the function produces is of accuracy d (i.e. it doesn't overestimate the shortest path from start to finish by more than d).

A given heuristic function $h(n)$ is *consistent* if the estimate is always less than or equal to the estimated distance between the goal n and any given neighbour, plus the estimated cost of reaching that neighbor:

$$c(n,m) + h(m) \geq h(n)$$

$c\ (n,m)$ being the distance between nodes n and m. Additionally, if $h(n)$ is consistent, then we know the optimal path to any node that has been already inspected. This means that this function is *optimal*.

2.4 Problem Reduction

In problem reduction, a complex problem is broken down or decomposed into a set of primitive sub problem; solutions for these primitive sub-problems are easily obtained. The solutions for all the sub problems collectively give the solution for the complex problem.

2.4.1 AND-OR GRAPHS (OR)

The AND-OR GRAPH (or tree) is useful for representing the solution of problems that can solved by decomposing them into a set of smaller problems, all of which must then be solved. This decomposition, or reduction, generates arcs that we call AND arcs. One AND arc may point to any number of successor nodes, all of which must be solved in order for the arc to point to a solution. Just as in an OR graph,

several arcs may emerge from a single node, indicating a variety of ways in which the original problem might be solved. This is why the structure is called not simply an AND-graph but rather an AND-OR graph (which also happens to be an AND-OR tree)

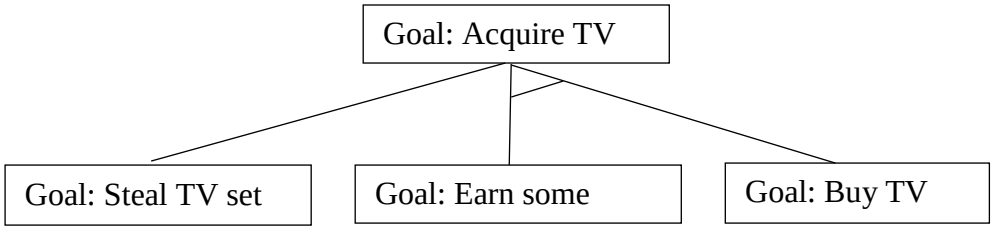


Figure 2.6 Simple AND/OR Graph

An algorithm to find a solution in an AND – OR graph must handle AND area appropriately. A* algorithm cannot search AND – OR graphs efficiently. This can be understanding from the give figure.

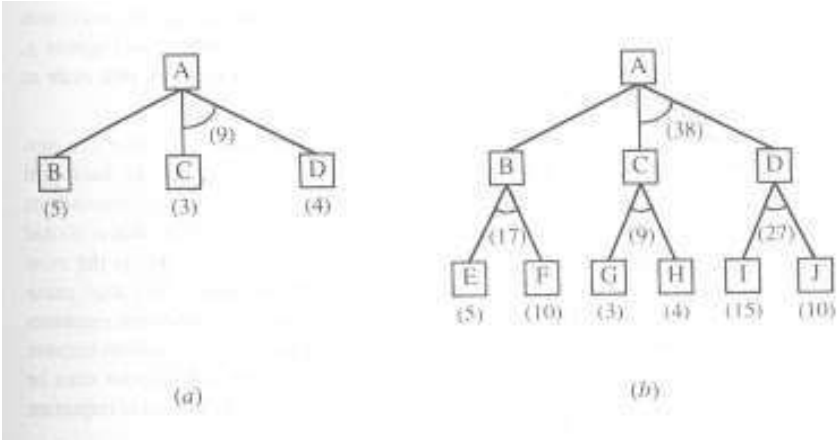


Figure 2.7: AND – OR graph

In figure (a) the top node A has been expanded producing two areas one leading to B and leading to C-D. the numbers at each node represent the value of f ‘ at that node (cost of getting to the goal state from current state). For simplicity, it is assumed that every operation (i.e. applying a rule) has unit cost, i.e., each are with single successor will have a cost of 1 and each of its components. With the available information till now, it appears that C is the most promising node to expand since its f ‘ = 3, the lowest but going through B would be better since to use C we must also use D’ and the cost would be 9(3+4+1+1). Through B it would be 6(5+1).

Thus the choice of the next node to expand depends not only n a value but also on whether that node is part of the current best path form the initial mode. Figure (b) makes this clearer. In figure the node G appears to be the most promising node, with the least f ‘ value. But G is not on the current beat path, since to use G we must use GH with a cost of 9 and again this demands that arcs be used (with a cost of 27). The path from A through B, E-F is better with a total cost of (17+1=18). Thus we can see that to search an AND-OR graph, the following three things must be done.

1. traverse the graph starting at the initial node and following the current best path, and accumulate the set of nodes that are on the path and have not yet been expanded.
2. Pick one of these unexpanded nodes and expand it. Add its successors to the graph and computer f ‘ (cost of the remaining distance) for each of them.
3. Change the f ‘ estimate of the newly expanded node to reflect the new information produced by its successors. Propagate this change backward through the graph. Decide which of the current best path.

The propagation of revised cost estimation backward is in the tree is not necessary in A* algorithm. This is because in AO* algorithm expanded nodes are re-examined so that the current best path can be selected. The working of AO* algorithm is illustrated in figure as follows:

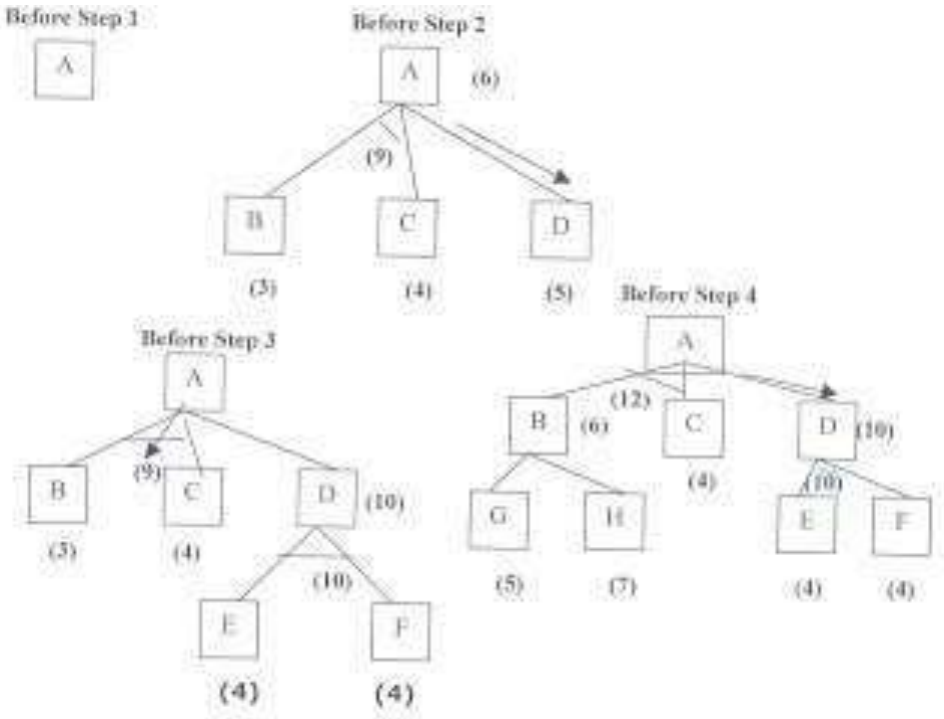


Figure 2.8: AND-OR Graph

Referring the figure. The initial node is expanded and D is Marked initially as promising node. D is expanded producing an AND arc E-F. f ' value of D is updated to 10. Going backwards we can see that the AND arc B-C is better. it is now marked as current best path. B and C have to be expanded next. This process continues until a solution is found or all paths have led to dead ends, indicating that there is no solution. An A* algorithm the path from one node to the other is always that of the lowest cost and it is independent of the paths through other nodes.

The algorithm for performing a heuristic search of an AND - OR graph is given below. Unlike A* algorithm which used two lists OPEN and CLOSED, the AO* algorithm uses a single structure G. G represents the part of the search graph generated so far. Each node in G points down to its immediate successors and up to its immediate predecessors, and also has with it the value of h' cost of a path from itself to a set of solution nodes. The cost of getting from the start nodes to the current node "g" is not stored as in the A* algorithm. This is because it is not possible to compute a single such value since there may be many paths to the same state. In AO* algorithm serves as the estimate of goodness of a node. Also a there should value called

FUTILITY is used. The estimated cost of a solution is greater than FUTILITY then the search is abandoned as too expansive to be practical.

For representing above graphs AO* algorithm is as follows

AO* ALGORITHM:

1. Let G consists only to the node representing the initial state call this node INTT. Compute h' (INIT).
2. Until INIT is labeled SOLVED or hi (INIT) becomes greater than FUTILITY, repeat the following procedure.

(I) Trace the marked arcs from INIT and select an unbounded node NODE.

(II) Generate the successors of NODE. if there are no successors then assign FUTILITY as h' (NODE). This means that NODE is not solvable. If there are successors, then for each one called SUCCESSOR, that is not also an ancestor of NODE do the following

(a) Add SUCCESSOR to graph G

(b) if successor is not a terminal node, mark it solved and assign zero to its h ' value.

(c) If successor is not a terminal node, compute it h' value.

(III) propagate the newly discovered information up the graph by doing the following. let S be a set of nodes that have been marked SOLVED. Initialize S to NODE. Until S is empty repeat the following procedure;

(a) select a node from S call if CURRENT and remove it from S.

(b) compute h' of each of the arcs emerging from CURRENT, Assign minimum h' to CURRENT.

(c) Mark the minimum cost path a s the best out of CURRENT.

(d) Mark CURRENT SOLVED if all of the nodes connected to it through the new marked are have been labeled SOLVED. Must

(e) If CURRENT has been marked SOLVED or its h ' has just changed, its new status be propagating backwards up the graph. hence all the ancestors of CURRENT are added to S.

(Refered from Artificial Intelligence TMH) AO* Search Procedure.

1. Place the start node on open.

2. Using the search tree, compute the most promising solution tree TP.
3. Select node n that is both on open and a part of tp, remove n from open and place it no closed.
4. If n is a goal node, label n as solved. If the start node is solved, exit with success where tp is the solution tree, remove all nodes from open with a solved ancestor.
5. If n is not solvable node, label n as unsolvable. If the start node is labeled as unsolvable, exit with failure. Remove all nodes from open, with unsolvable ancestors.
6. Otherwise, expand node n generating all of its successor compute the cost of for each newly generated node and place all such nodes on open.
7. Go back to step (2)

Note: AO* will always find minimum cost solution.

2.5 CONSTRAINTS SATISFACTION

Constraint satisfaction is a search procedure that operates in a space of constraint sets. The initial state contains the constraints that are originally given in the problem description. A goal state is any state that has been constrained “enough” where "enough” must be defined for each problem. For example, for crypt arithmetic, enough means that each letter has been assigned a unique numeric value.

Constraint satisfaction is a two-step process: -

1. Constraint are discovered and propagated as far as possible throughout the system. Then, if there is still not a solution, search begins. A guess about something is made and added as a new constraint. Propagation can then occur with this new constraint, and so forth. Propagation arises from the fact that there are usually dependencies among the constraints. These dependencies occur because many constraints involve more than one object and many objects participate in more than one constraint. So, for example, assume we start with one constraint, N=E + 1. Then. if we added the constraint N = 3, we could propagate that to get a stronger constraint on E, namely l that E = 2. Constraint propagation also arises from the presence of inference rules that allow additional constraints to be inferred from the ones that are given. Constraint

propagation terminates for one of two reasons. First, a contradiction may be detected. If this happens, then there is no solution consistent with all the known constraints. If the contradiction involves only those constraints that were given as part of the problem specification (as opposed to ones that were guessed during problem solving), then no solution exists. The second possible reason for termination is that the propagation has run out of steam and there are no further changes that can be made on the basis of current knowledge. If this happens and a solution has not yet been adequately specified, then search is necessary to get the process moving again.

2. After we have achieved all that we proceed to the second step where some hypothesis about a way to strengthen the constraints must be made. In the case of the crypt arithmetic problem, for example, this usually means guessing a particular value for some letter. Once this has been done, constraint propagation can begin again from this new state. If a solution is found, it can be reported. If still guesses are required, they can be made. If a contradiction is detected, then backtracking can be used to try a different guess and proceed with it.

Many problems in AI can be considered as problems of constraint satisfaction, in which the goal state satisfies a given set of constraint. constraint satisfaction problems can be solved by using any of the search strategies. The general form of the constraint satisfaction procedure is as follows:

Until a complete solution is found or until all paths have led to lead ends, do

1. select an unexpanded node of the search graph.
2. Apply the constraint inference rules to the selected node to generate all possible new constraints.
3. If the set of constraints contains a contradiction, then report that this path is a dead end.
4. If the set of constraints describes a complete solution, then report success.
5. If neither a constraint nor a complete solution has been found, then apply the rules to generate new partial solutions. Insert these partial solutions into the search graph.

Example: consider the crypt arithmetic problems.

SEND

+ MORE

MONEY

Assign decimal digit to each of the letters in such a way that the answer to the problem is correct to the same letter occurs more than once, it must be assign the same digit each time. no two different letters may be assigned the same digit. Consider the crypt arithmetic problem.

SEND

+ MORE

MONEY

CONSTRAINTS: -

1. no two digits can be assigned to same letter.
2. only single digit number can be assign to a letter.
3. no two letters can be assigned same digit.
4. Assumption can be made at various levels such that they do not contradict each other.
5. The problem can be decomposed into secured constraints. A constraint satisfaction approach may be used.
6. Any of search techniques may be used.
7. Backtracking may be performed as applicable us applied search techniques.
8. Rule of arithmetic may be followed.

Initial state of problem.

D=? E=? Y=? N=? R=? O=? S=? M=? C1=? C2=?

C1, C2, C3 stands for the carry variables respectively.

Goal State: the digits to the letters must be assigned in such a manner so that the sum is satisfied.

Solution Process:

We are following the depth-first method to solve the problem.

1. initial guess m=1 because the sum of two single digits can generate at most a carry '1'.

2. When n=1 o=0 or 1 because the largest single digit number added to m=1 can generate the sum of either 0 or 1 depend on the carry received from the carry sum. By this we conclude that o=0 because m is already 1 hence we cannot assign same digit another letter (rule no.)
3. We have m=1 and o=0 to get o=0 we have s=8 or 9, again depending on the carry received from the earlier sum.

The same process can be repeated further. The problem has to be composed into various constraints. And each constraint is to be satisfied by guessing the possible digits that the letters can be assumed that the initial guess has been already made. rest of the process is being shown in the form of a tree, using depth-first search for the clear understandability of the solution process.

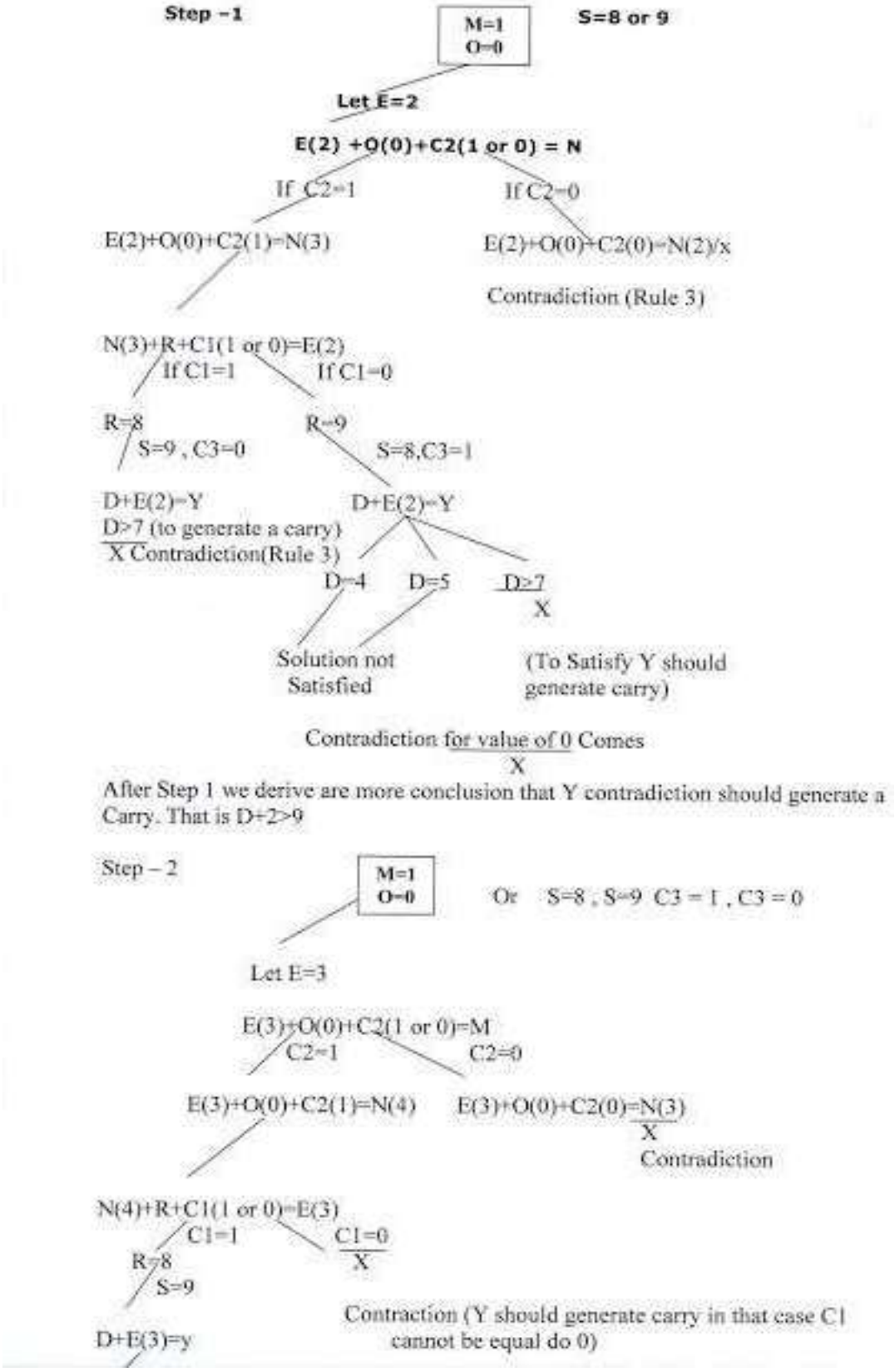
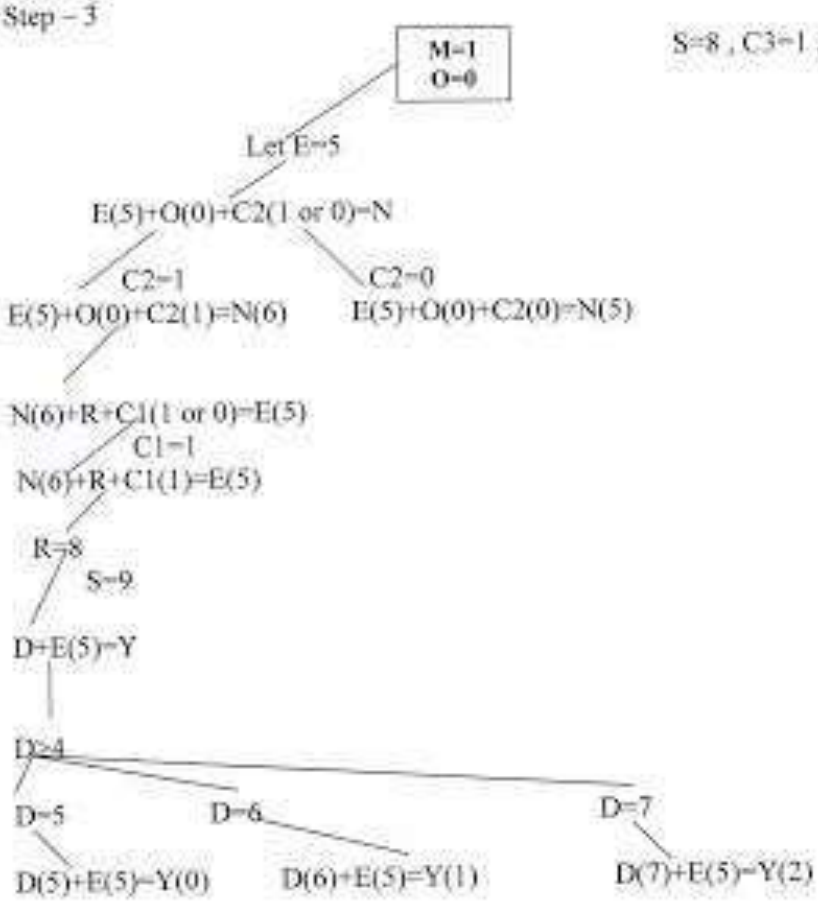


Figure 2.9: D>6(Controduction)

After Step 2 , we found that C1 cannot be Zero, Since Y has to generate a carry to satisfy goal state. From this step onwards, no need to branch for C1=0.

Step – 3



At Step (4) we have assigned a single digit to every letter in accordance with the constraints & production rules.

Now by backtracking , we find the different digits assigned to different letters and hence reach the solution state.

Solution State:-

Y = 2
D = 7
S = 9
R = 8
N = 6
E = 5
O = 0
M = 1
C1 = 1
C2 = 0
C3 = 0

	C3(0)	C2(1)	C1(1)	
	S(9)	E(5)	N(6)	D(7)
+	M(1)	O(0)	R(8)	E(5)
M(1)	O(0)	N(6)	E(5)	Y(2)

Figure 2.10: Example

2.6 MEANS END ANALYSIS

The means-ends analysis process centers around the detection of differences between the current state and the goal state. The means-ends analysis process can then be applied recursively to the sub problem of the main problem. In order to focus the system's attention on the big problems first, the differences can be assigned priority levels. Differences of higher priority can then be considered before lower priority ones.

Means-ends analysis relies on a set of rules that can transform one problem state into another. These rules are usually not represented with complete state descriptions on each side.

Algorithm: Means-Ends Analysis (CURRENT, GOAL)

1. Compare CURRENT with GOAL. If there are no differences between them then return.
2. Otherwise, select the most important difference and reduce it doing the following until success or failure is signaled:
    a. Select an as yet untried operator O that is applicable to the current difference. If there are no such operators, then signal failure.
    b. Attempt to apply O to CURRENT. Generate descriptions of two states: O-START, a state in which O's preconditions are satisfied and O-RESULT, the state that would result if O were applied in O-START.
    c. If (FIRST-PART ← MEA (CURRENT, O-START)) and (LAST-PART ← MEA (O-RESULT, GOAL)) are successful, then signal success and return the result of concatenating FIRST-PART,O, and LAST-PART.

In particular, the order in which differences are considered can be critical. It is important that significant differences be reduced before less critical ones. If this is not done, a great deal of effort may be wasted on situations that take care of themselves once the main parts of the problem are solved. The simple process we have described is usually not adequate for solving complex problems. The number of permutations of

differences may get too large; Working on one difference may interfere with the plan for reducing another.

2.7 MINIMAX SEARCH PROCEDURE

The minimax search is a depth-first and depth limited procedure. The idea is to start at the current position and use the plausible-move generator to generate the set of possible successor positions. Now we can apply the static evolution function to those positions and simply choose the best one. After doing so, we can back that value up to the starting position to represent our evolution of it. Here we assume that static evolution function returns larger values to indicate good situations for us. So our goal is to maximize the value of the static evaluation function of the next board position.
The opponents' goal is to minimize the value of the static evaluation function.

The alternation of maximizing and minimizing at alternate ply when evaluations are to be pushed back up corresponds to the opposing strategies of the two players is called MINIMAX.

It is the recursive procedure that depends on two procedures

- MOVEGEN (position, player)— The plausible-move generator, which returns a list of nodes representing the moves that can make by Player in Position.
- STATIC (position, player)– static evaluation function, which returns a number representing the goodness of Position from the standpoint of Player.

With any recursive program, we need to decide when recursive procedure should stop. There are the variety of factors that may influence the decision they are,

- Has one side won?
- How many plies have we already explored? Or how much time is left?
- How stable is the configuration?

We use DEEP-ENOUGH which assumed to evaluate all of these factors and to return TRUE if the search should be stopped at the current level and FALSE otherwise.
It takes two parameters, position, and depth, it will ignore its position parameter and simply return TRUE if its depth parameter exceeds a constant cut off value.
One problem that arises in defining MINIMAX as a recursive procedure is that it needs to return not one but two results.

- The backed-up value of the path it chooses.
- The path itself. We return the entire path even though probably only the first element, representing the best move from the current position, actually needed.

We assume that MINIMAX returns a structure containing both results and we have two functions, VALUE and PATH that extract the separate components.

Initially, it takes three parameters, a board position, the current depth of the search, and the player to move,

- MINIMAX (current,0, player-one) If player –one is to move
- MINIMAX (current,0, player-two) If player –two is to move

2.7.1 Adding alpha-beta cutoffs

Minimax procedure is a depth-first process. One path is explored as far as time allows, the static evolution function is applied to the game positions at the last step of the path. The efficiency of the depth-first search can improve by branch and bound technique in which partial solutions that clearly worse than known solutions can abandon early. It is necessary to modify the branch and bound strategy to include two bounds, one for each of the players. This modified strategy called alpha-beta pruning. It requires maintaining of two threshold values, one representing a lower bound on that a maximizing node may ultimately assign (we call this alpha). And another representing an upper bound on the value that a minimizing node may assign (this we call beta). Each level must receive both the values, one to use and one to pass down to the next level to use. The MINIMAX procedure as it stands does not need to treat maximizing and minimizing levels differently. Since it simply negates evaluation each time it changes levels. Instead of referring to alpha and beta, MINIMAX uses two values, USE-THRESH and PASSTHRESH. USE-THRESH used to compute cutoffs. PASS-THRESH passed to next level as its USETHRESH. USE-THRESH must also pass to the next level, but it will pass as PASS-THRESH so that it can be passed to the third level down as USE-THRESH again, and so forth. Just as values had to negate each time they passed across levels.

Still, there is no difference between the code required at maximizing levels and that required at minimizing levels. PASS-THRESH should always the maximum of the

value it inherits from above and the best move found at its level. If PASS-THRESH updated the new value should propagate both down to lower levels. And back up to higher ones so that it always reflects the best move found anywhere in the tree. The MINIMAX-A-B requires five arguments, position, depth, player, Use-thresh, and pass Thresh. MINIMAX-A-B (current,0, player-one, maximum value static can compute, minimum value static can compute). Iterative Deepening Search(IDS) or Iterative Deepening Depth First Search(IDDFS)

There are two common ways to traverse a graph, BFS and DFS. Considering a Tree (or Graph) of huge height and width, both BFS and DFS are not very efficient due to following reasons.

1. **DFS** first travers's nodes going through one adjacent of root, then next adjacent. The problem with this approach is, if there is a node close to root, but not in first few subtrees explored by DFS, then DFS reaches that node very late. Also, DFS may not find shortest path to a node (in terms of number of edges).

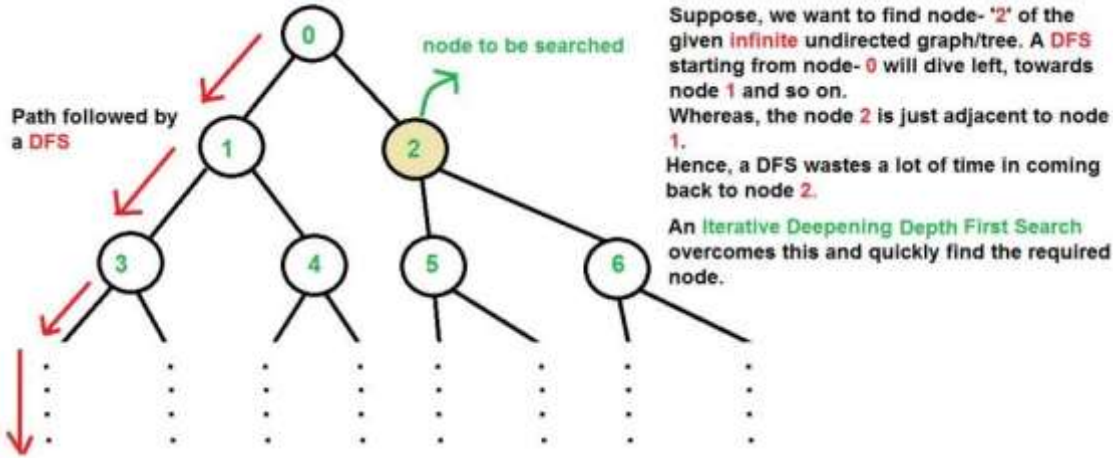


Figure 2.11: DFS Working

2. **BFS** goes level by level, but requires more space. The space required by DFS is O(d) where d is depth of tree, but space required by BFS is O(n) where n is number of nodes in tree (Why? Note that the last level of tree can have around n/2 nodes and second last level n/4 nodes and in BFS we need to have every level one by one in queue).

IDDFS combines depth-first search's space-efficiency and breadth-first search's fast search (for nodes closer to root).

How does IDDFS work?

IDDFS calls DFS for different depths starting from an initial value. In every call, DFS is restricted from going beyond given depth. So basically we do DFS in a BFS fashion.

Algorithm:

```
// Returns true if target is reachable from
// src within max_depth
bool IDDFS(src, target, max_depth)
for limit from 0 to max_depth
        if DLS(src, target, limit) == true return true
                return false
        bool DLS(src, target, limit)
        if (src == target)
                return true;
// If reached the maximum depth,
// stop recursing.
if (limit <= 0)
        return false;
foreach adjacent i of src
        if DLS(i, target, limit?1)
                return true
return false
```

An important thing to note is, we visit top level nodes multiple times. The last (or max depth) level is visited once, second last level is visited twice, and so on. It may seem expensive, but it turns out to be not so costly, since in a tree most of the nodes are in the bottom level. So it does not matter much if the upper levels are visited multiple times.

2.8 SUMMARY

In this lesson we have discussed the most common methods of problem representation in AI are:

- State Space Representation.
- Problem Reduction.

State Space Representation is highly beneficial in AI because they provide all possible states, operators and the goals. In case of problem reduction, a complex problem is broken down or decomposed into a set of primitive sub problem; solutions for these primitive sub-problems are easily obtained.

Search is a characteristic of almost all AI problems. Search strategies can be compared by their time and space complexities. It is important to determine the complexity of a given strategy before investing too much programming effort, since many search problems are in traceable.

In case of brute search (Uninformed Search or Blind Search) , nodes in the space are explored mechanically until a goal is found, a time limit has been reached, or failure occurs. Examples of brute force search are breadth first search and death first search. In case of Heuristic Search (Informed Search) cost or another function is used to select the most promising path at each point in the search. Heuristics evolution functions are used in the best first strategy to find good solution paths. A solution is not always guaranteed with this type of search, but in most practical cases, good or acceptable solutions are often found.

2.9 KEY WORDS

State Space Representation, Problem Reduction, Depth First Search, Breadth First Search, Hill Climbing, Branch & Bound, Best First Search, Constraints Satisfactions & Mean End Analysis.

2.10 CHECK YOUR PROGRESS

Q1. Discuss various types of problem representation. Also discuss their advantages & disadvantages.

Q2. What are various heuristics search techniques? Explain how they are different from the search techniques.

Q3. What do you understand by uniformed search? What are its advantages & disadvantages over informed search? What is breadth first search better than depth first search better than depth first and vice-versa? Explain.

Q4. Differentiate between following: -

(a) Local maximum and plateau in hill climbing search.

(b) Depth first search and breadth first search.

Q5. Write sort notes on the following: -

(a) Production System

(b) Constraints Satisfaction

(c) Mean End Analysis

2.11 REFERENCE/SUGGESTED READING

- Foundations of Artificial Intelligence and Expert System - V S Janakiraman, K Sarukesi, & P Gopalakrishanan, Macmillan Series.
- Artificial Intelligence – E. Rich and K. Knight
- Principles of Artificial Intelligence – Nilsson
- Expert Systems-Paul Harmon and David King, Wiley Press.
- Rule Based Expert System-Bruce G. Buchanan and Edward H. Shortliffe, eds., Addison Wesley.
- Introduction to Artificial Intelligence and Expert System- Dan W. Patterson, PHI, Feb., 2003.

SUBJECT: ARTIFICIAL INTELLIGENCE	
COURSE CODE: MCA-23	**AUTHOR:**
LESSON NO. 3	**VETTER:**
KNOWLEDGE REPRESENTATION	

UNIT STRUCTURE

3.0 Objectives

3.1 Knowledge representation

3.2 Using Predicate Logic

3.3 Representing Instance and ISA Relationships

3.4 Computable Functions and Predicates

3.5 Propositional Resolution

3.6 Unification

3.7 Resolution in predicate logic

3.8 Summary

3.9 Check Your Progress

3.10 Reference/Suggested Reading

3.0 OBJECTIVE

This lesson is providing an introduction about logic and knowledge representation techniques. The logic is used to represent knowledge. Various knowledge representation schemes are also discussed in detail. Upon the completion of this lesson students are able to learn how to represent AI problem(s) with the help of knowledge representation schemes.

3.1 KNOWLEDGE REPRESENTATION

For the purpose of solving complex problems c\encountered in AI, we need both a large amount of knowledge and some mechanism for manipulating that knowledge to create solutions to new problems. A variety of ways of representing knowledge

(facts) have been exploited in AI programs. In all variety of knowledge representations, we deal with two kinds of entities.

A. Facts: Truths in some relevant world. These are the things we want to represent.
B. Representations of facts in some chosen formalism. these are things we will actually be able to manipulate.

One way to think of structuring these entities is at two levels: (a) the knowledge level, at which facts are described, and (b) the symbol level, at which representations of objects at the knowledge level are defined in terms of symbols that can be manipulated by programs.

The facts and representations are linked with two-way mappings. This link is called representation mappings. The forward representation mapping maps from facts to representations. The backward representation mapping goes the other way, from representations to facts.

One common representation is natural language (particularly English) sentences. Regardless of the representation for facts we use in a program, we may also need to be concerned with an English representation of those facts in order to facilitate getting information into and out of the system. We need mapping functions from English sentences to the representation we actually use and from it back to sentences.

Representations and Mappings

- In order to solve complex problems encountered in artificial intelligence, one needs both a large amount of knowledge and some mechanism for manipulating that knowledge to create solutions.
- Knowledge and Representation are two distinct entities. They play central but distinguishable roles in the intelligent system.
- Knowledge is a description of the world. It determines a system's competence by what it knows.
- Moreover, Representation is the way knowledge is encoded. It defines a system's performance in doing something.
- Different types of knowledge require different kinds of representation.

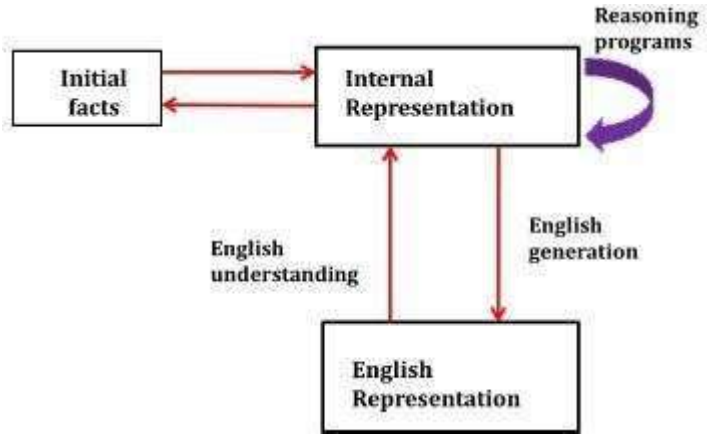


Figure 3.1: Mapping between Facts and Representations

The Knowledge Representation models/mechanisms are often based on:

- Logic
- Rules
- Frames
- Semantic Net

Knowledge is categorized into two major types:

1. Tacit corresponds to “informal” or “implicit “
   - Exists within a human being;
   - It is embodied.
   - Difficult to articulate formally.
   - Difficult to communicate or share.
   - Moreover, Hard to steal or copy.
   - Drawn from experience, action, subjective insight
2. Explicit formal type of knowledge, Explicit
   - Explicit knowledge
   - Exists outside a human being;
   - It is embedded.
   - Can be articulated formally.
   - Also, Can be shared, copied, processed and stored.
   - So, Easy to steal or copy

- Drawn from the artifact of some type as a principle, procedure, process, concepts. A variety of ways of representing knowledge have been exploited in AI programs.

There are two different kinds of entities, we are dealing with.

1. Facts: Truth in some relevant world. Things we want to represent.
2. Also, Representation of facts in some chosen formalism. Things we will actually be able to manipulate.

These entities structured at two levels:

1. The knowledge level, at which facts described.
2. Moreover, the symbol level, at which representation of objects defined in terms of symbols that can manipulate by programs

Framework of Knowledge Representation

- The computer requires a well-defined problem description to process and provide a well- defined acceptable solution.
- Moreover, to collect fragments of knowledge we need first to formulate a description in our spoken language and then represent it in formal language so that computer can understand.
- Also, the computer can then use an algorithm to compute an answer. So, this process illustrated as,

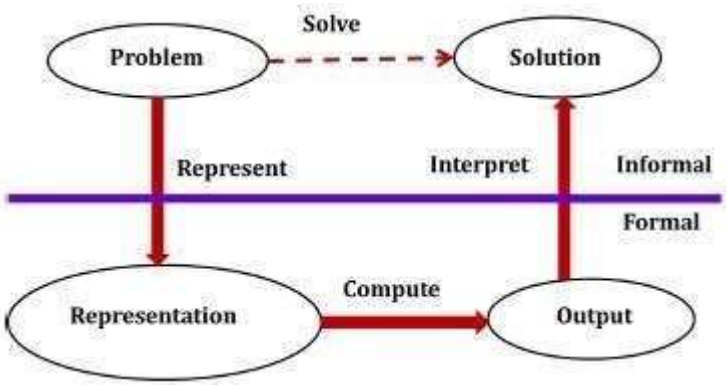


Figure 3.2: **Knowledge Representation Framework**

The steps are:

- The informal formalism of the problem takes place first.
- It then represented formally and the computer produces an output.
- This output can then have represented in an informally described solution that user understands or checks for consistency.

The Problem solving requires,

- Formal knowledge representation, and
- Moreover, Conversion of informal knowledge to a formal knowledge that is the conversion of implicit knowledge to explicit knowledge.

Mapping between Facts and Representation

- Knowledge is a collection of facts from some domain.
- Also, we need a representation of "facts "that can manipulate by a program.
- Moreover, Normal English is insufficient, too hard currently for a computer program to draw inferences in natural languages.
- Thus some symbolic representation is necessary.

A good knowledge representation enables fast and accurate access to knowledge and understanding of the content.

A knowledge representation system should have following properties.

1. Representational Adequacy
   - The ability to represent all kinds of knowledge that are needed in that domain.
2. Inferential Adequacy
   - Also, the ability to manipulate the representational structures to derive new structures corresponding to new knowledge inferred from old.
3. Inferential Efficiency
   - The ability to incorporate additional information into the knowledge structure that can be used to focus the attention of the inference mechanisms in the most promising direction.
4. Acquisitional Efficiency

- Moreover, the ability to acquire new knowledge using automatic methods wherever possible rather than reliance on human intervention.

Knowledge Representation Schemes Relational Knowledge

- The simplest way to represent declarative facts is a set of relations of the same sort used in the database system.
- Provides a framework to compare two objects based on equivalent attributes. o Any instance in which two different objects are compared is a relational type of knowledge.
- The table below shows a simple way to store facts.
- Also, the facts about a set of objects are put systematically in columns.
- This representation provides little opportunity for inference.

Player	Height	Weight	Bats - Throws
Aaron	6-0	180	Right - Right
Mays	5-10	170	Right - Right
Ruth	6-2	215	Left - Left
Williams	6-3	205	Left - Right

Figure 3.3: Data of Player

- Given the facts, it is not possible to answer a simple question such as: "Who is the heaviest player?"
- Also, but if a procedure for finding the heaviest player is provided, then these facts will enable that procedure to compute an answer.
- Moreover, we can ask things like who "bats – left" and "throws – right".

Inheritable Knowledge

- Here the knowledge elements inherit attributes from their parents.
- The knowledge embodied in the design hierarchies found in the functional, physical and process domains.
- Within the hierarchy, elements inherit attributes from their parents, but in many cases, not all attributes of the parent elements prescribed to the child elements.
- Also, the inheritance is a powerful form of inference, but not adequate.
- Moreover, the basic KR (Knowledge Representation) needs to augment with inference mechanism.

- Property inheritance: The objects or elements of specific classes inherit attributes and values from more general classes.
- So, the classes organized in a generalized hierarchy.

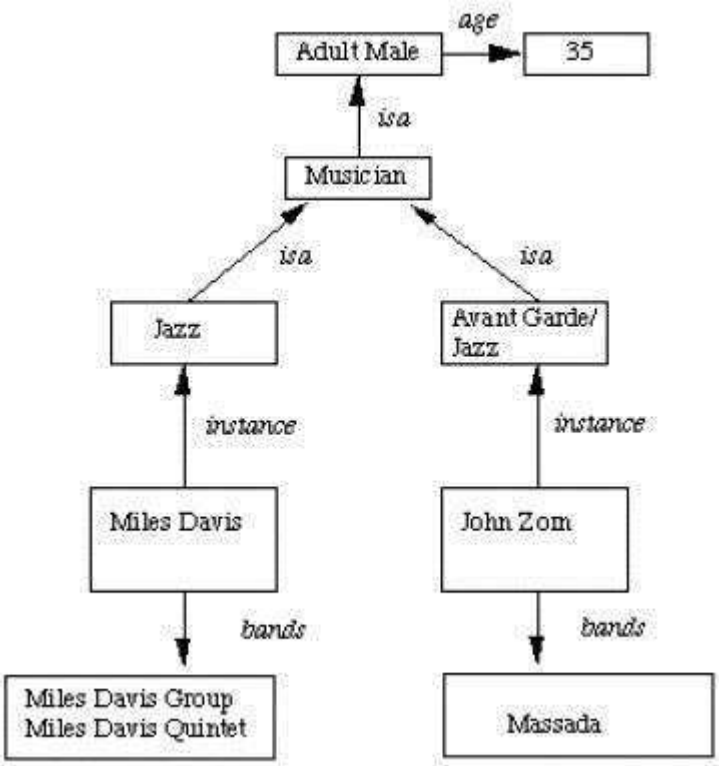


Figure 3.4: Sturcture

- Boxed nodes — objects and values of attributes of objects.
- Arrows — the point from object to its value.
- This structure is known as a slot and filler structure, semantic network or a collection of frames.

The steps to retrieve a value for an attribute of an instance object:

1. Find the object in the knowledge base
2. If there is a value for the attribute report it
3. Otherwise look for a value of an instance, if none fail
4. Also, go to that node and find a value for the attribute and then report it
5. Otherwise, search through using is until a value is found for the attribute.

Inferential Knowledge

- This knowledge generates new information from the given information.
- This new information does not require further data gathering form source but does require analysis of the given information to generate new knowledge.
- Example: given a set of relations and values, one may infer other values or relations. A predicate logic (a mathematical deduction) used to infer from a set of attributes. Moreover, Inference through predicate logic uses a set of logical operations to relate individual data.
- Represent knowledge as formal logic: All dogs have tails $\forall x: dog(x) \rightarrow hastail(x)$
- Advantages:
- A set of strict rules.
- Can use to derive more facts.
- Also, Truths of new statements can be verified.
- Guaranteed correctness.
- So, many inference procedures available to implement standard rules of logic popular in AI systems. e.g Automated theorem proving.

Procedural Knowledge

- A representation in which the control information, to use the knowledge, embedded in the knowledge itself. For example, computer programs, directions, and recipes; these indicate specific use or implementation;
- Moreover, Knowledge encoded in some procedures, small programs that know how to do specific things, how to proceed.
- Advantages:
- Heuristic or domain-specific knowledge can represent.
- Moreover, Extended logical inferences, such as default reasoning facilitated.
- Also, Side effects of actions may model. Some rules may become false in time. Keeping track of this in large systems may be tricky.
- Disadvantages:
- Completeness — not all cases may represent.
- Consistency — not all deductions may be correct. e.g If we know that Fred is a bird we might deduce that Fred can fly. Later we might discover that Fred is an emu.

- Modularity sacrificed. Changes in knowledge base might have far-reaching effects.
- Cumbersome control information.

3.2 USING PREDICATE LOGIC

Representation of Simple Facts in Logic

Propositional logic is useful because it is simple to deal with and a decision procedure for it exists.

Also, In order to draw conclusions, facts are represented in a more convenient way as,

1. Marcus is a man.
   - man(Marcus)
2. Plato is a man.
   - man(Plato)
3. All men are mortal.
   - mortal(men)

But propositional logic fails to capture the relationship between an individual being a man and that individual being a mortal.

- How can these sentences be represented so that we can infer the third sentence from the first two?
- Also, Propositional logic commits only to the existence of facts that may or may not be the case in the world being represented.
- Moreover, It has a simple syntax and simple semantics. It suffices to illustrate the process of inference.
- Propositional logic quickly becomes impractical, even for very small worlds.

Predicate logic

First-order Predicate logic (FOPL) models the world in terms of

- Objects, which are things with individual identities
- Properties of objects that distinguish them from other objects
- Relations that hold among sets of objects

- Functions, which are a subset of relations where there is only one “value” for any given “input”

First-order Predicate logic (FOPL) provides

- Constants: a, b, dog33. Name a specific object.
- Variables: X, Y. Refer to an object without naming it.
- Functions: Mapping from objects to objects.
- Terms: Refer to objects
- Atomic Sentences: in(dad-of(X), food6) Can be true or false, Correspond to propositional symbols P, Q.

A well-formed formula (*wff*) is a sentence containing no “free” variables. So, that is, all variables are “bound” by universal or existential quantifiers.

(∀x) P (x, y) has x bound as a universally quantified variable, but y is free.

Quantifiers

Universal quantification

- (∀x)P(x) means that P holds for all values of x in the domain associated with that variable
- E.g., (∀x) dolphin(x) → mammal(x) Existential quantification
- (∃ x)P(x) means that P holds for some value of x in the domain associated with that variable
- E.g., (∃ x) mammal(x) ∧ lays-eggs(x)

Also, Consider the following example that shows the use of predicate logic as a way of representing knowledge.

1. Marcus was a man.
2. Marcus was a Pompeian.
3. All Pompeians were Romans.
4. Caesar was a ruler.
5. Also, All Pompeians 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 well-formed formulas (*wffs*) 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 Pompeians were either loyal to Caesar or hated him.
   - inclusive-or
   - ∀x: Roman(x) → loyalto(x, Caesar) ∨ hate(x, Caesar)
   - exclusive-or
   - ∀x: Roman(x) → (loyalto(x, Caesar) ∧¬ hate(x, Caesar)) ∨
   - (¬loyalto(x, Caesar) ∧ hate(x, Caesar))
6. Everyone is loyal to someone.
   - ∀x: ∃y: loyalto(x, y)
7. People only try to assassinate rulers they are not loyal to.
   - ∀x: ∀y: person(x) ∧ ruler(y) ∧ tryassassinate(x, y)
   - →¬loyalto(x, y)
8. Marcus tried to assassinate Caesar.
   - tryassassinate(Marcus, Caesar)

Now suppose if we want to use these statements to answer the question: **Was Marcus loyal to Caesar?**

Also, Now let's try to produce a formal proof, reasoning backward from the desired goal: ¬ Ioyalto(Marcus, Caesar)

In order to prove the goal, we need to use the rules of inference to transform it into another goal (or possibly a set of goals) that can, in turn, transformed, and so on, until there are no unsatisfied goals remaining.

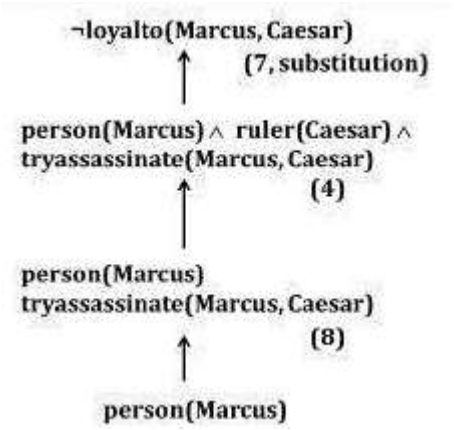


Figure 3.5: An attempt to prove ¬loyalto(Marcus, Caesar).

- The problem is that, although we know that Marcus was a man, we do not have any way to conclude from that that Marcus was a person. Also, we need to add the representation of another fact to our system, namely***: ∀man(x) → person(x)***
- Now we can satisfy the last goal and produce a proof that Marcus was not loyal to Caesar.
- Moreover, from this simple 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:

1. Many English sentences are ambiguous (for example, 5, 6, and 7 above). Choosing the correct interpretation may be difficult.
2. Also, there is often a choice of how to represent the knowledge. Simple representations are desirable, but they may exclude certain kinds of reasoning.
3. Similarly, even in very simple situations, a set of sentences is unlikely to contain all the information necessary to reason about the topic at hand. In order to be able to use a set of statements effectively. Moreover, it is usually necessary to have access to another set of statements that represent facts that people consider too obvious to mention.

3.3 Representing Instance and ISA Relationships

- Specific attributes **instance** and **isa** play an important role particularly in a useful form of reasoning called property inheritance.
- The predicates instance and isa explicitly captured the relationships they used to express, namely class membership and class inclusion.

- 4.2 shows the first 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 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 the figure contains representations that use the ***instance*** predicate explicitly.

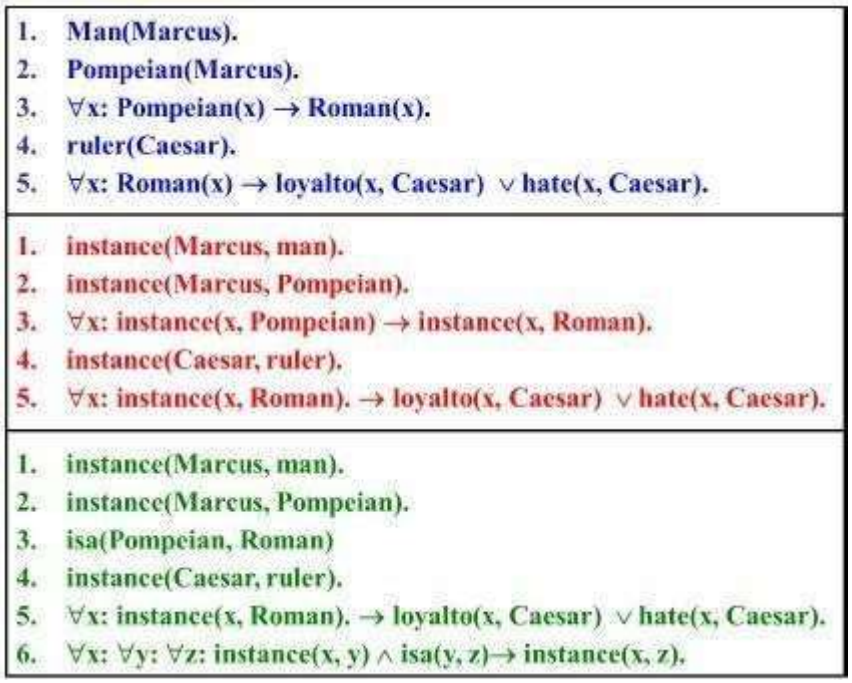


Figure 3.6: **Three ways of representing class membership: ISA Relationships**

- 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, described as shown in sentence 3.
- The implication rule states that if an object is an instance of the subclass Pompeian then it is an instance of the superclass Roman.
- Note that this rule is equivalent to the standard set-theoretic definition of the subclass- superclass relationship.

- The third part contains representations that use both the ***instance*** and ***isa*** predicates explicitly.
- The use of the ***isa*** predicate simplifies the representation of sentence 3, but it requires that one additional axiom (shown here as number 6) be provided.

3.4 Computable Functions and Predicates

- To express simple facts, such as the following greater-than and less-than relationships: gt (1, O) It (0,1) gt (2,1) It (1,2) gt (3,2) It (2,3)
- It is often also useful to have computable functions as well as computable predicates. Thus we might want to be able to evaluate the truth of gt(2 + 3,1)
- To do so requires that we first compute the value of the plus function given the arguments 2 and 3, and then send the arguments 5 and 1 to gt.

Consider the following set of facts, again involving Marcus:

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 men are mortal.

   x: man(x) → mortal(x)
5. All Pompeians died when the volcano erupted in 79 A.D. erupted (volcano, 79) ∧ ∀ x: [Pompeian(x) → died (x, 79)]
6. No mortal lives longer than 150 years.

   x: t1: At2: mortal(x) born (x, t1) gt(t2 – t1,150) → died(x, t2)
7. It is now 1991.

   now = 1991

So, above example shows how these ideas of computable functions and predicates can be useful. It also makes use of the notion of equality and allows equal objects to be substituted for each other whenever it appears helpful to do so during a proof.

- So, now suppose we want to answer the question “Is Marcus alive?”

- The statements suggested here, there may be two ways of deducing an answer.
- Either we can show that Marcus is dead because he was killed by the volcano or we can show that he must be dead because he would otherwise be more than 150 years old, which we know is not possible.
- Also, as soon as we attempt to follow either of those paths rigorously, however, we discover, just as we did in the last example, that we need some additional knowledge. 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.

   x: t: [alive (x, t) → ¬ dead (x, t)] [¬ dead (x, t) → alive (x, t)]

9. If someone dies, then he is dead at all later times. x: t1: At2*: died (x, t1) gt(t2, t1) → dead(x, t2)*

So, now let's attempt to answer the question "Is Marcus alive?" by proving: ¬ ***alive (Marcus, now)***

3.5 Propositional Resolution

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 step 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 the 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 adding it to the set of classes available to the procedure.

3.6 Unification Algorithm

- In propositional logic, it is easy to determine that two literals cannot both be true at the same time.
- 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 example, man(John) and ¬man(John) is a contradiction, while the man(John) and ¬man(Spot) 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.
- There is a straightforward recursive procedure, called the unification algorithm, that does it.

Algorithm: Unify (L1, L2)

1. If L1 or 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. Also, 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 LI and L2 have a different number of arguments, then return {FAIL}.
4. Set SUBST to NIL. (At the end of this procedure, SUBST will contain all the substitutions used to unify L1 and L2.)
5. For I ← 1 to the number of arguments in L1:
    a. Call Unify with the i^{th} argument of L1 and the i^{th} argument of L2, putting the 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.

3.7 RESOLUTION IN PREDICATE LOGIC

We can now state the resolution algorithm for predicate logic as follows, assuming a set of given statements F and a statement to be proved P:

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 a contradiction found, no progress can make, or a predetermined amount of effort has expanded.
    a. Select two clauses. Call these the parent clauses.
    b. Resolve them together. The resolvent will the disjunction of all 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 T2 and the other contains T1 and if T1 and T2 are unifiable, then neither T1 nor T2 should appear in the resolvent. We call T1 and 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 omit from the resolvent.
    c. If the resolvent is an empty clause, then a contradiction has found. Moreover, if it is not, then adding it to the set of classes available to the procedure.

RESOLUTION PROCEDURE

- Resolution is a procedure, which gains its efficiency from the fact that it operates on statements that have been converted to a very convenient standard form.
- Resolution produces proofs by refutation.
- In other words, ***to prove a statement (i.e., to show that it is valid), resolution attempts to show that the negation of the statement produces a contradiction with the known statements (i.e., that it is unsatisfiable).***

- The resolution procedure is a simple iterative process: at each step, two clauses, called the parent clauses, are compared (resolved), resulting in a new clause that has inferred from them. The new clause represents ways that the two parent clauses interact with each other. Suppose that there are two clauses in the system:

  winter V summer

  ¬ winter V cold
- Now we observe that precisely one of winter and ¬ winter will be true at any point.
- If winter is true, then cold must be true to guarantee the truth of the second clause. If ¬ winter is true, then summer must be true to guarantee the truth of the first clause.
- Thus we see that from these two clauses we can deduce ***summer V cold***
- This is the deduction that the resolution procedure will make.
- Resolution operates by taking two clauses that each contains the same literal, in this example, ***winter***.
- Moreover, the literal must occur in the positive form in one clause and in negative form in the other. The resolvent obtained by combining all of the literals of the two parent clauses except the ones that cancel.
- If the clause that produced is the empty clause, then a contradiction has found.

For example, the two clauses winter

¬ winter

will produce the empty clause.

NATURAL DEDUCTION USING RULES

Testing whether a proposition is a tautology by testing every possible truth assignment is expensive—there are exponentially many. We need a **deductive system**, which will allow us to construct proofs of tautologies in a step-by-step fashion.

The system we will use is known as **natural deduction**. The system consists of a set of **rules of inference** for deriving consequences from premises. One builds a proof tree whose root is the proposition to be proved and whose leaves are the initial

assumptions or axioms (for proof trees, we usually draw the root at the bottom and the leaves at the top).

For example, one rule of our system is known as **modus ponens**. Intuitively, this says that if we know P is true, and we know that P implies Q, then we can conclude Q.

$$\frac{P \qquad P \Rightarrow Q}{Q} \text{ (modus ponens)}$$

The propositions above the line are called **premises**; the proposition below the line is the **conclusion**. Both the premises and the conclusion may contain metavariables (in this case, P and Q) representing arbitrary propositions. When an inference rule is used as part of a proof, the metavariables are replaced in a consistent way with the appropriate kind of object (in this case, propositions).

Most rules come in one of two flavors: **introduction** or **elimination** rules. Introduction rules introduce the use of a logical operator, and elimination rules eliminate it. Modus ponens is an elimination rule for ⇒. On the right-hand side of a rule, we often write the name of the rule. This is helpful when reading proofs. In this case, we have written (modus ponens). We could also have written (⇒-elim) to indicate that this is the elimination rule for ⇒.

Rules for Conjunction

Conjunction (∧) has an introduction rule and two elimination rules:

$$\frac{P \qquad Q}{P \wedge Q} \text{ (∧-intro)} \qquad \frac{P \wedge Q}{P} \text{ (∧-elim-left)} \qquad \frac{P \wedge Q}{Q} \text{ (∧-elim-right)}$$

Rule for T

The simplest introduction rule is the one for T. It is called "unit". Because it has no premises, this rule is an **axiom**: something that can start a proof.

$$\frac{}{T} \text{ (unit)}$$

Rules for Implication

In natural deduction, to prove an implication of the form P ⇒ Q, we assume P, then reason under that assumption to try to derive Q. If we are successful, then we can conclude that P ⇒ Q.

In a proof, we are always allowed to introduce a new assumption P, then reason under that assumption. We must give the assumption a name; we have used the name x in the example below. Each distinct assumption must have a different name.

$$\frac{}{[x : P]}\ (\text{assum})$$

Because it has no premises, this rule can also start a proof. It can be used as if the proposition P were proved. The name of the assumption is also indicated here.

However, you do not get to make assumptions for free! To get a complete proof, all assumptions must be eventually **discharged.** This is done in the implication introduction rule. This rule introduces an implication $P \Rightarrow Q$ by discharging a prior assumption [x : P]. Intuitively, if Q can be proved under the assumption P, then the implication $P \Rightarrow Q$ holds without any assumptions. We write x in the rule name to show which assumption is discharged. This rule and modus ponens are the introduction and elimination rules for implications.

$$\frac{\begin{array}{c}[x : P]\\ \vdots\\ Q\end{array}}{P \Rightarrow Q}\ (\Rightarrow\text{-intro/x}) \qquad \frac{P \quad P \Rightarrow Q}{Q}\ (\Rightarrow\text{-elim, modus ponens})$$

A proof is valid only if every assumption is eventually discharged. This must happen in the proof tree below the assumption. The same assumption can be used more than once.

Rules for Disjunction

$$\frac{P}{P \vee Q}\ (\vee\text{-intro-left}) \qquad \frac{Q}{P \vee Q}\ (\vee\text{-intro-right}) \qquad \frac{P \vee Q \quad P \Rightarrow R \quad Q \Rightarrow R}{R}\ (\vee\text{-elim})$$

Rules for Negation

A negation $\neg P$ can be considered an abbreviation for $P \Rightarrow \bot$:

$$\frac{P \Rightarrow \bot}{\neg P}\ (\neg\text{-intro}) \qquad \frac{\neg P}{P \Rightarrow \bot}\ (\neg\text{-elim})$$

Rules for Falsity

$$\frac{\begin{array}{c}[x : \neg P] \\ \vdots \\ \bot\end{array}}{P}\ \text{(reductio ad absurdum, RAA/x)} \qquad \frac{\bot}{P}\ \text{(ex falso quodlibet, EFQ)}$$

Reductio ad absurdum (RAA) is an interesting rule. It embodies proofs by contradiction. It says that if by assuming that P is false we can derive a contradiction, then P must be true. The assumption x is discharged in the application of this rule. This rule is present in classical logic but not in **intuitionistic** (constructive) logic. In intuitionistic logic, a proposition is not considered true simply because its negation is false.

Excluded Middle

Another classical tautology that is not intuitionistically valid is the **law of the excluded middle**, P ∨¬P. We will take it as an axiom in our system. The Latin name for this rule is *tertium non datur*, but we will call it *magic*.

$$\frac{}{P \vee \neg P}\ \text{(magic)}$$

Proofs

A proof of proposition P in natural deduction starts from axioms and assumptions and derives P with all assumptions discharged. Every step in the proof is an instance of an inference rule with metavariables substituted consistently with expressions of the appropriate syntactic class.

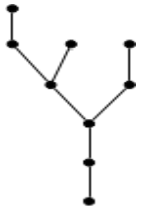
Example

For example, here is a proof of the proposition (A ⇒ B ⇒ C) ⇒ (A ∧ B ⇒ C).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[y : A \wedge B]}\,(\text{A})}{A}\,(\wedge\text{E}) \quad \dfrac{}{[x : A \Rightarrow B \Rightarrow C]}\,(\text{A})}{B \Rightarrow C}\,(\Rightarrow\text{E}) \quad \dfrac{}{[y : A \wedge B]}\,(\text{A})}{B}\,(\wedge\text{E})}{C}\,(\Rightarrow\text{E})}{A \wedge B \Rightarrow C}\,(\Rightarrow\text{I,y})}{(A \Rightarrow B \Rightarrow C) \Rightarrow (A \wedge B \Rightarrow C)}\,(\Rightarrow\text{I,x})$$

The final step in the proof is to derive $(A \Rightarrow B \Rightarrow C) \Rightarrow (A \wedge B \Rightarrow C)$ from $(A \wedge B \Rightarrow C)$, which is done using the rule ($\Rightarrow$-intro), discharging the assumption $[x : A \Rightarrow B \Rightarrow C]$. To see how this rule generates the proof step, substitute for the metavariables P, Q, x in the rule as follows: $P = (A \Rightarrow B \Rightarrow C)$, $Q = (A \wedge B \Rightarrow C)$, and $x = x$. The immediately previous step uses the same rule, but with a different substitution: $P = A \wedge B$, $Q = C$, $x = y$.

The proof tree for this example has the following form, with the proved proposition at



the root and axioms and assumptions at the leaves.

A proposition that has a complete proof in a deductive system is called a **theorem** of that system.

Soundness and Completeness

A measure of a deductive system's power is whether it is powerful enough to prove all true statements. A deductive system is said to be **complete** if all true statements are theorems (have proofs in the system). For propositional logic and natural deduction, this means that all tautologies must have natural deduction proofs. Conversely, a deductive system is

called **sound** if all theorems are true. The proof rules we have given above are in fact sound and complete for propositional logic: every theorem is a tautology, and every tautology is a theorem. Finding a proof for a given tautology can be difficult. But once the proof is found, checking that it is indeed a proof is completely mechanical, requiring no intelligence or insight whatsoever. It is therefore a very strong argument that the thing proved is in fact true.

We can also make writing proofs less tedious by adding more rules that provide reasoning shortcuts. These rules are sound if there is a way to convert a proof using them into a proof using the original rules. Such added rules are called **admissible**.

3.8 Summary

We have considered prepositional and predicate logics in this lesson as knowledge representation schemes. We have learned that Predicate Logic has sound theoretical foundation; it is not expressive enough for many practical problems. FOPL, on the other provides a theoretically sound basis and permits great latitude of expressiveness. In FOPL one can easily code object descriptions and relations among objects as well as general assertions about classes of similar objects.

- Modus Pones is a property of prepositions that is useful in resolution and can be represented as $P \text{ and } P \rightarrow Q \Rightarrow Q$ where P and Q are two clauses.
- Resolution produces proofs by refutation.

Finally, rules, a subset of FOPL, were described as a popular representation scheme.

3.9 Key Words

Predicate Logic, FOPL, Modus Ponen, Unification, Resolution & Dependency Directed Backtracking.

3.10 Check Your Progress

Answer the following Questions:

Q1. What are the limitations of logic as representation scheme?

Q2. Differentiate between Propositional & Predicate Logic.

Q3. Perform resolution on the set of clauses

A: P V Q V R B: `P V R C: `Q Q: `R

Q4. Write short notes on the following:

a. Unification

b. Modus Ponen

c. Directed Backtracking

d. Resolution

3.11 Reference/Suggested Reading

- Foundations of Artificial Intelligence and Expert System - V S Janakiraman, K Sarukesi, & P Gopalakrishanan, Macmillan Series.

- Artificial Intelligence – E. Rich and K. Knight
- Principles of Artificial Intelligence – Nilsson
- Expert Systems-Paul Harmon and David King, Wiley Press.
- Rule Based Expert System-Bruce G. Buchanan and Edward H. Shortliffe, eds., Addison Wesley.
- Introduction to Artificial Intelligence and Expert System- Dan W. Patterson, PHI, Feb., 2003.

SUBJECT: ARTIFICIAL INTELLIGENCE	
COURSE CODE: MCA-23	**AUTHOR:**
LESSON NO. 4	**VETTER:**
RULE BASED SYSTEM	

UNIT STRUCTURE

4.0 Objectives

4.1 Procedural vs Declarative Knowledge

4.2 Forward vs Backward Reasoning

4.3 Conflict Resolution

4.4 Forward Chaining System

4.5 Backward Chaining System

4.6 Use of No Backtrack

4.7 Summary

4.8 Check Your Progress

4.9 Reference/Suggested Reading

4.0 Objective

The objective of this lesson is to provide an overview of rule-based system. This lesson discusses about procedural versus declarative knowledge. Students are come to know how to handle the problems, related with forward and backward chaining. Upon completion of this lesson, students are able to solve their problems using rule-based system.

4.1 Introduction

Using a set of assertions, which collectively form the 'working memory', and a set of rules that specify how to act on the assertion set, a rule-based system can be created. Rule-based systems are fairly simplistic, consisting of little more than a set of if-then statements, but provide the basis for so-called "expert systems" which are widely used in many fields. The concept of an expert system is this: the knowledge of an expert is

encoded into the rule set. When exposed to the same data, the expert system AI will perform in a similar manner to the expert.

Rule-based systems are a relatively simple model that can be adapted to any number of problems. As with any AI, a rule-based system has its strengths as well as limitations that must be considered before deciding if it's the right technique to use for a given problem. Overall, rule-based systems are really only feasible for problems for which any and all knowledge in the problem area can be written in the form of if-then rules and for which this problem area is not large. If there are too many rules, the system can become difficult to maintain and can suffer a performance hit.

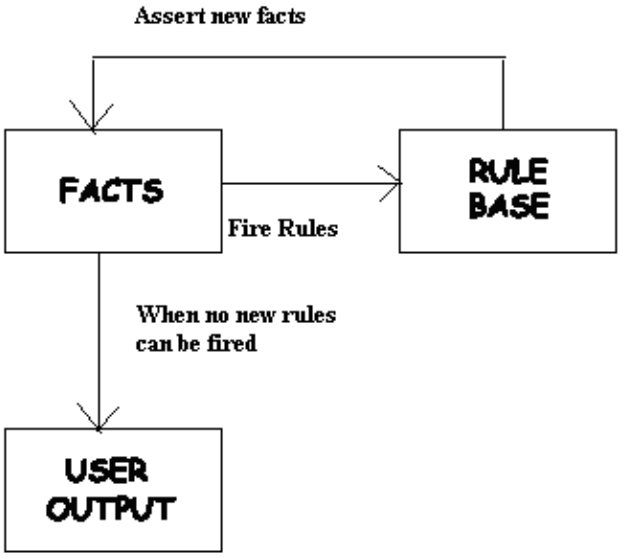
To create a rule-based system for a given problem, you must have (or create) the following:

1. A set of facts to represent the initial working memory. This should be anything relevant to the beginning state of the system.
2. A set of rules. This should encompass any and all actions that should be taken within the scope of a problem, but nothing irrelevant. The number of rules in the system can affect its performance, so you don't want any that aren't needed.
3. A condition that determines that a solution has been found or that none exists. This is necessary to terminate some rule-based systems that find themselves in infinite loops otherwise.

Theory of Rule-Based Systems

The rule-based system itself uses a simple technique: It starts with a rule-base, which contains all of the appropriate knowledge encoded into If-Then rules, and a working memory, which may or may not initially contain any data, assertions or initially known information. The system examines all the rule conditions (IF) and determines a subset, the conflict set, of the rules whose conditions are satisfied based on the working memory. Of this conflict set, one of those rules is triggered (fired). Which one is chosen is based on a conflict resolution strategy. When the rule is fired, any actions specified in its THEN clause are carried out. These actions can modify the working memory, the rule-base itself, or do just about anything else the system programmer decides to include. This loop of firing rules and performing actions continues until one of two conditions are met: there are no more rules whose

conditions are satisfied or a rule is fired whose action specifies the program should terminate.



Which rule is chosen to fire is a function of the conflict resolution strategy. Which strategy is chosen can be determined by the problem or it may be a matter of preference. In any case, it is vital as it controls which of the applicable rules are fired and thus how the entire system behaves. There are several different strategies, but here are a few of the most common:

- **First Applicable**: If the rules are in a specified order, firing the first applicable one allows control over the order in which rules fire. This is the simplest strategy and has a potential for a large problem: that of an infinite loop on the same rule. If the working memory remains the same, as does the rule-base, then the conditions of the first rule have not changed and it will fire again and again. To solve this, it is a common practice to suspend a fired rule and prevent it from re-firing until the data in working memory, that satisfied the rule's conditions, has changed.
- **Random**: Though it doesn't provide the predictability or control of the first-applicable strategy, it does have its advantages. For one thing, its unpredictability is an advantage in some circumstances (such as games for example). A random strategy simply chooses a single random rule to fire from the conflict set. Another possibility for a random strategy is a fuzzy rule-based system in which each of the rules has a probability such that some rules are more likely to fire than others.

- **Most Specific**: This strategy is based on the number of conditions of the rules. From the conflict set, the rule with the most conditions is chosen. This is based on the assumption that if it has the most conditions then it has the most relevance to the existing data.
- **Least Recently Used**: Each of the rules is accompanied by a time or step stamp, which marks the last time it was used. This maximizes the number of individual rules that are fired at least once. If all rules are needed for the solution of a given problem, this is a perfect strategy.
- **"Best" rule**: For this to work, each rule is given a 'weight,' which specifies how much it should be considered over the alternatives. The rule with the most preferable outcomes is chosen based on this weight.

There are two broad kinds of rule system: *forward chaining* systems, and *backward chaining* systems. In a forward chaining system you start with the initial facts, and keep using the rules to draw new conclusions (or take certain actions) given those facts. In a backward chaining system you start with some hypothesis (or goal) you are trying to prove, and keep looking for rules that would allow you to conclude that hypothesis, perhaps setting new sub goals to prove as you go. Forward chaining systems are primarily data-driven, while backward chaining systems are goal-driven.

Procedural versus Declarative Knowledge

We have discussed various search techniques in previous units. Now we would consider a set of rules that represent,

1. Knowledge about relationships in the world and
2. Knowledge about how to solve the problem using the content of the rules.

Procedural vs Declarative Knowledge Procedural Knowledge

- A representation in which the control information that is necessary to use the knowledge is embedded in the knowledge itself for e.g. computer programs, directions, and recipes; these indicate specific use or implementation;
- The real difference between declarative and procedural views of knowledge lies in where control information resides.

For example, consider the following

Man (Marcus) Man (Caesar) Person (Cleopatra)

∀x: Man(x) → Person(x)

Now, try to answer the question.? Person(y)

The knowledge base justifies any of the following answers.

Y=Marcus Y=Caesar Y=Cleopatra

- We get more than one value that satisfies the predicate.
- If only one value needed, then the answer to the question will depend on the order in which the assertions examined during the search for a response.
- If the assertions declarative, then they do not themselves say anything about how they will be examined. In case of procedural representation, they say how they will examine.

Declarative Knowledge

- A statement in which knowledge specified, but the use to which that knowledge is to be put is not given.
- For example, laws, people's name; these are the facts which can stand alone, not dependent on other knowledge;
- So to use declarative representation, we must have a program that explains what is to do with the knowledge and how.
- For example, a set of logical assertions can combine with a resolution theorem prover to give a complete program for solving problems but in some cases, the logical assertions can view as a program rather than data to a program.
- Hence the implication statements define the legitimate reasoning paths and automatic assertions provide the starting points of those paths.
- These paths define the execution paths which is similar to the 'if then else "in traditional programming.
- So logical assertions can view as a procedural representation of knowledge.

Logic Programming – Representing Knowledge Using Rules

- Logic programming is a programming paradigm in which logical assertions viewed as programs.
- These are several logic programming systems; PROLOG is one of them.

- ***A PROLOG program consists of several logical assertions where each is a horn clause***

i.e. a clause with at most one positive literal.

- Ex: P, P V Q, P → Q
- The facts are represented on Horn Clause for two reasons.
  1. Because of a uniform representation, a simple and efficient interpreter can write.
  2. The logic of Horn Clause decidable.
- Also, the first two differences are the fact that PROLOG programs are actually sets of Horn clause that have been transformed as follows: -
  1. If the Horn Clause contains no negative literal, then leave it as it is.
  2. Also, otherwise rewrite the Horn clauses as an implication, combining all of the negative literals into the antecedent of the implications and the single positive literal into the consequent.
- Moreover, this procedure causes a clause which originally consisted of a disjunction of literals (one of them was positive) to be transformed into a single implication whose antecedent is a conjunction universally quantified.
- But when we apply this transformation, any variables that occurred in negative literals and so now occur in the antecedent become existentially quantified, while the variables in the consequent are still universally quantified.

For example, the PROLOG clause P(x): – Q (x, y) is equal to logical expression ∀x: ∃y: Q (x, → P(x).

- The difference between the logic and PROLOG representation is that the PROLOG interpretation has a fixed control strategy. And so, the assertions in the PROLOG program define a particular search path to answer any question.
- But, the logical assertions define only the set of answers but not about how to choose among those answers if there is more than one.

Consider the following example:

1. Logical representation

   $\forall x : pet(x) \wedge small(x) \rightarrow apartmentpet(x)$

   $\forall x : cat(x) \wedge dog(x) \rightarrow pet(x)$

   $\forall x : poodle(x) \rightarrow dog(x) \wedge small(x)\ poodle(fluffy)$

2. Prolog representation

apartmentpet (x) : pet(x), small (x) pet (x): cat (x)
pet (x): dog(x)
dog(x): poodle (x) small (x): poodle(x) poodle (fluffy)

Preliminaries of Rule-based systems may be viewed as use of logical assertions within the knowledge representation.

A declarative representation is one in which knowledge is specified, but the use to which that knowledge is to be put is not given. A declarative representation, we must augment it with a program that specifies what is to be done to the knowledge and how. For example, a set of logical assertions can be combined with a resolution theorem prover to give a complete program for solving problems. There is a different way, though, in which logical assertions can be viewed, namely as a program, rather than as data to a program. In this view, the implication statements define the legitimate reasoning paths and the atomic assertions provide the starting points (or, if we reason backward, the ending points) of those paths.

A procedural representation is one in which the control information that is necessary to use the knowledge is considered to be embedded in the knowledge itself. To use a procedural representation, we need to augment it with an interpreter that follows the instructions given in the knowledge.

Screening logical assertions as code is not a very essential idea, given that all programs are really data to other programs that interpret (or compile) and execute them. The real difference between the declarative and the procedural views of knowledge lies in where control information resides. For example, consider the knowledge base:

man (Marcus)
man (Caesar)
person(Cleopatra)
$\forall x$: man (x)$\rightarrow$ person(x)

Now consider trying to extract from this knowledge base the answer to the question

$\exists y$: person(y)

We want to bind y to a particular value for which person is true. Our knowledge base justifies any of the following answers:

y = Marcus

y = Caesar

y = Cleopatra

For the reason that there is more than one value that satisfies the predicate, but only one value is needed, the answer to the question will depend on the order in which the assertions are examined during the search for a response.

Of course, nondeterministic programs are possible. So, we could view these assertions as a nondeterministic program whose output is simply not defined. If we do this, then we have a "procedural" representation that actually contains no more information than does the "declarative" form. But most systems that view knowledge as procedural do not do this. The reason for this is that, at least if the procedure is to execute on any sequential or on most existing parallel machines, some decision must be made about the order in which the assertions will be examined. There is no hardware support for randomness. So if the interpreter must have a way of-deciding, there is no real reason not to specify it as part of the definition of the language and thus to define the meaning of any particular program in the language. For example, we might specify that assertions will be examined in the order in which they appear in the program and that search will proceed depth-first, by which we mean that if a new subgoal is established then it will be pursued immediately and other paths will only be examined if the new one fails. If we do that, then the assertions we gave above describe a program that will answer our question with

y = Cleopatra

To see clearly the difference between declarative and procedural representations, consider the following assertions:

man(Marcus)

man(Caesar)

$\forall x : man(x) \rightarrow person(x)$

person(Cleopatra)

Viewed declaratively, this is the same knowledge base that we had before. All the same answers are supported by the system and no one of them is explicitly selected. But viewed procedurally, and using the control model we used to get Cleopatra as our answer before, this is a different knowledge base since now the answer to our question is Marcus. This happens because the first statement that can achieve the person goal is the inference rule

$\forall x$: man(x) → person(x).

This rule sets up a subgoal to find a man. Again the statements are examined from the beginning, and now Marcus is found to satisfy the subgoal and thus also the goal. So Marcus is reported as the answer.

It is important to keep in mind that although we have said that a procedural representation encodes control information in the knowledge base, it does so only to the extent that the interpreter for the knowledge base recognizes that control information. So we could have gotten a different answer to the person question by leaving our original knowledge base intact and changing the interpreter so that it examines statements from last to first (but still pursuing depth-first search). Following this control regime, we report Caesar as our answer.

There has been a great deal of disagreement in AI over whether declarative or procedural knowledge representation frameworks are better. There is no clear-cut answer to the question. As you can see from this discussion, the distinction between the two forms is often very fuzzy. Rather than try to answer the question of which approach is better, what we do in the rest of this chapter is to describe ways in which rule formalisms and interpreters can be combined to solve problems. We begin with a mechanism called logic programming, and then we consider more flexible structures for rule-based systems.

4.2 Forwards versus Backwards Reasoning

A search procedure must find a path between initial and goal states. There are two directions in which a search process could proceed. The two types of search are:

1. Forward search which starts from the start state
2. Backward search that starts from the goal state

The production system views the forward and backward as symmetric processes. Consider a game of playing 8 puzzles. The rules defined are

Square 1 empty and square 2 contains tile n. →

- *Also, Square 2 empty and square 1 contains the tile n.*

Square 1 empty Square 4 contains tile n. →

- *Also, Square 4 empty and Square 1 contains tile n.*

We can solve the problem in 2 ways:

1. Reason forward from the initial state
   - Step 1. Begin building a tree of move sequences by starting with the initial configuration at the root of the tree.
   - Step 2. Generate the next level of the tree by finding all rules ***whose left-hand side matches*** against the root node. The right-hand side is used to create new configurations.
   - Step 3. Generate the next level by considering the nodes in the previous level and applying it to all rules whose left-hand side match.
2. Reasoning backward from the goal states:
   - Step 1. Begin building a tree of move sequences by starting with the goal node configuration at the root of the tree.
   - Step 2. Generate the next level of the tree by finding all rules ***whose right-hand side matches*** against the root node. The left-hand side used to create new configurations.
   - Step 3. Generate the next level by considering the nodes in the previous level and applying it to all rules whose right-hand side match.
   - So, The same rules can use in both cases.
   - Also, In forwarding reasoning, the left-hand sides of the rules matched against the current state and right sides used to generate the new state.
   - Moreover, In backward reasoning, the right-hand sides of the rules matched against the current state and left sides are used to generate the new state.

There are four factors influencing the type of reasoning. They are,

1. Are there more possible start or goal state? We move from smaller set of sets to the length.
2. In what direction is the branching factor greater? We proceed in the direction with the lower branching factor.
3. Will the program be asked to justify its reasoning process to a user? If, so then it is selected since it is very close to the way in which the user thinks.
4. What kind of event is going to trigger a problem-solving episode? If it is the arrival of a new factor, the forward reasoning makes sense. If it is a query to which a response is desired, backward reasoning is more natural.

Example 1 of Forward versus Backward Reasoning

- It is easier to drive from an unfamiliar place from home, rather than from home to an unfamiliar place. Also, If you consider a home as starting place an unfamiliar place as a goal then we have to backtrack from unfamiliar place to home.

Example 2 of Forward versus Backward Reasoning

- Consider a problem of symbolic integration. Moreover, The problem space is a set of formulas, which contains integral expressions. Here START is equal to the given formula with some integrals. GOAL is equivalent to the expression of the formula without any integral. Here we start from the formula with some integrals and proceed to an integral free expression rather than starting from an integral free expression.

Example 3 of Forward versus Backward Reasoning

- The third factor is nothing but deciding whether the reasoning process can justify its reasoning. If it justifies, then it can apply. For example, doctors are usually unwilling to accept any advice from diagnostics process because it cannot explain its reasoning.

Example 4 of Forward versus Backward Reasoning

- Prolog is an example of backward chaining rule system. In Prolog rules restricted to Horn clauses. This allows for rapid indexing because all the rules for deducing a given fact share the same rule head. Rules matched with unification procedure. Unification tries to find a set of bindings for variables to equate a sub-goal with the

head of some rule. Rules in the Prolog program matched in the order in which they appear.

Combining Forward and Backward Reasoning

- Instead of searching either forward or backward, you can search both simultaneously.
- Also, That is, start forward from a starting state and backward from a goal state simultaneously until the paths meet.
- This strategy called Bi-directional search. The following figure shows the reason for a Bidirectional search to be ineffective.

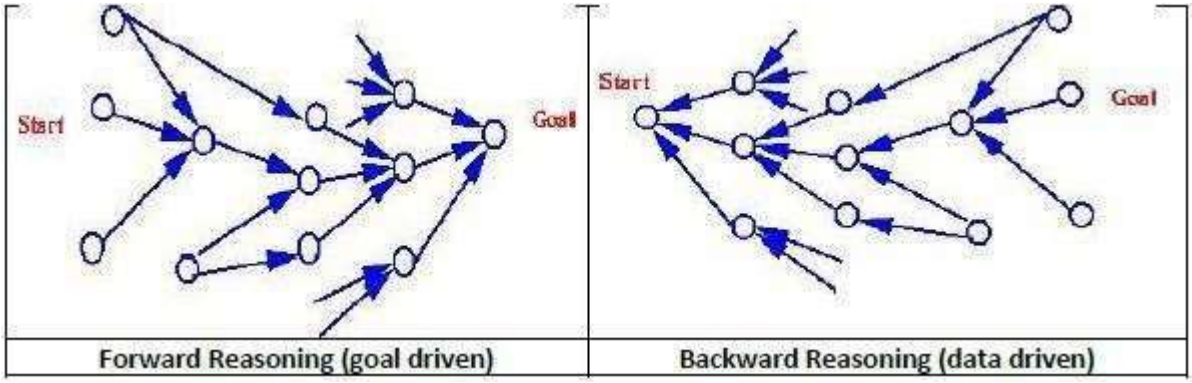


Figure 3.7: Forward versus Backward Reasoning

- Also, the two searches may pass each other resulting in more work.
- Based on the form of the rules one can decide whether the same rules can apply to both forward and backward reasoning.
- Moreover, if left-hand side and right of the rule contain pure assertions then the rule can reverse.
- And so the same rule can apply to both types of reasoning.
- If the right side of the rule contains an arbitrary procedure, then the rule cannot reverse.

So, in this case, while writing the rule the commitment to a direction of reasoning must make.

Whether you use forward or backwards reasoning to solve a problem depends on the properties of your rule set and initial facts. Sometimes, if you have some particular goal (to test some hypothesis), then backward chaining will be much more efficient,

as you avoid drawing conclusions from irrelevant facts. However, sometimes backward chaining can be very wasteful - there may be many possible ways of trying to prove something, and you may have to try almost all of them before you find one that works. Forward chaining may be better if you have lots of things you want to prove (or if you just want to find out in general what new facts are true); when you have a small set of initial facts; and when there tend to be lots of different rules which allow you to draw the same conclusion. Backward chaining may be better if you are trying to prove a single fact, given a large set of initial facts, and where, if you used forward chaining, lots of rules would be eligible to fire in any cycle. The guidelines forward & backward reasoning are as follows:

- Move from the smaller set of states to the larger set of states.
- Proceed in the direction with the lower branching factor.
- Proceed in the direction that corresponds more closely with the way the user will think.
- Proceed in the direction that corresponds more closely with the way the problem-solving episodes will be triggered.
- Forward rules: to encode knowledge about how to respond to certain input.
- Backward rules: to encode knowledge about how to achieve particular goals.

Problems in AI can be handled in two of the available ways:

- Forward, from the start states
- Backward, from the goal states, which is used in PROLOG as well.

Taking into account the problem of solving a particular instance of the 8-puzzle. The rules to be used for solving the puzzle can be written as shown in Figure 4.1.

Reason forward from the initial states. Begin building a tree of move sequences that might be solutions by starting with the initial configuration(s) at the root of the tree. Generate the next

Assume the areas of the tray are numbered:

1	2	3
4	5	6
7	8	9

Square 1 empty and Square 2 contains tile *n* →
Square 2 empty and Square 1 contains tile *n*
Square 1 empty and Square 4 contains tile *n* →
Square 4 empty and Square 1 contains tile *n*
Square 2 empty and Square 1 contains tile *n* →
Square 1 empty and Square 2 contains tile *n*

Figure 4.1: A Sample of the Rules for solving the 8-Puzzle

level of the tree by finding all the rules whose left sides match the root node and using their right sides to create the new configurations. Generate the next level by taking each node generated at the previous level and applying to it all of the rules whose left sides match it. Continue until a configuration that matches the goal state is generated.

Reason backward from the goal states. Begin building a tree of move sequences that might be solutions by starting with the goal configuration(s) at the root of the tree. Generate the next level of the tree by finding all the rules whose right sides match the root node. These are all the rules that, if only we could apply them, would generate the state we want. Use the left sides of the rules to generate the nodes at this second level of the tree. Generate the next level of the tree by taking each node at the previous level and finding all the rules whose right sides match it. Then use the corresponding left sides to generate the new nodes. Continue until a node that matches the initial state is generated. This method of reasoning backward from the desired final state is often called goal-directed reasoning.

To reason forward, the left sides are matched against the current state and the right sides (the results) are used to generate new nodes until the goal is reached. To reason backward, the right sides are matched against the current node and the left sides are used to generate new nodes representing new goal states to be achieved. This continues until one of these goal states is matched by an initial state.

In the case of the 8-puzzle, it does not make much difference whether we reason, forward or backward; about the same number of paths will be explored in either case. But this is not always true. Depending on the topology of the problem space, it may be significantly more efficient to search in one direction rather than the other. Four factors influence the question of whether it is better to reason forward or backward:

- Are there more possible start states or goal states? We would like to move from the smaller set of states to the larger (and thus easier to find) set of states.

- In which direction is the branching factor (i.e., the average number of nodes that can be reached directly from a single node)? We would like to proceed in the direction with the lower branching factor.
- Will the program be asked to justify its reasoning process to a user? If so, it is important to proceed in the direction that corresponds more closely with the way the user will think.
- What kind of event is going to trigger a problem-solving episode? If it is the arrival of a new fact, forward reasoning makes sense. If it is a query to which a response is desired, backward reasoning is more natural.

We may as well consider a few practical examples that make these issues clearer. Have you ever noticed that it seems easier to drive from an unfamiliar place home than from home to an unfamiliar place? The branching factor is roughly the same in both directions. But for the purpose of finding our way around, there are many more locations that count as being home than there are locations that count as the unfamiliar target place. Any place from which we know how to get home can be considered as equivalent to home. If we can get to any such place, we can get home easily. But in order to find a route from where we are to an unfamiliar place, we pretty much have to be already at the unfamiliar place. So in going toward the unfamiliar place, we are aiming at a much smaller target than in going home. This suggests that if our starting position is home and our goal position is the unfamiliar place, we should plan our route by reasoning backward from the unfamiliar place.

On the other hand, consider the problem of symbolic integration. The problem space is the set of formulas, some of which contain integral expressions. The start state is a particular formula containing some integral expression. The desired goal state is a formula that is equivalent to the initial one and that does not contain any integral expressions. So we begin with a single easily identified start state and a huge number of possible goal states. Thus to solve this problem, it is better to reason forward using the rules for integration to try to generate an integral-free expression than to start with arbitrary integral-free expressions, use the rules for differentiation, and try to generate the particular integral we are trying to solve. Again we want to head toward the largest target; this time that means chaining forward. These two examples have

illustrated the importance of the relative number of start states to goal states in determining the optimal direction in which to search when the branching factor is approximately the same in both directions. When the branching factor is not the same, however, it must also be taken into account.

Consider again the problem of proving theorems in some particular domain of mathematics. Our goal state is the particular theorem to be proved. Our initial states are normally a small set of axioms. Neither of these sets is significantly bigger than the other. But consider the branching factor in each of the two directions, from a small set of axioms we can derive a very large number of theorems. On the other hand, this large number of theorems must go back to the small set of axioms. So the branching factor is significantly greater going forward from the axioms to the theorems than it is going backward from theorems to axioms. This suggests that it would be much better to reason backward when trying to prove theorems. Mathematicians have long realized this, as have the designers of theorem-proving programs.

The third factor that determines the direction in which search should proceed is the need to generate coherent justifications of the reasoning process as it proceeds. This is often crucial for the acceptance of programs for the performance of very important tasks. For example, doctors are unwilling to accept the advice of a diagnostic program that cannot explain its reasoning to the doctors' satisfaction. This issue was of concern to the designers of MYCIN, a program that diagnoses infectious diseases. It reasons backward from its goal of determining the cause of a patient's illness. To do that, it uses rules that tell it such things as "If the organism has the following set of characteristics as determined by the lab results, then it is likely that it is organism x. By reasoning backward using such rules, the program can answer questions like "Why should I perform that test you just asked for?" with such answers as "Because it would help to determine whether organism x is present." By describing the search process as the application of a set of production rules, it is easy to describe the specific search algorithms without reference to the direction of the search.

We can also search both forward from the start state and backward from the goal simultaneously until two paths meet somewhere in between. This strategy is called bidirectional search. It seems appealing if the number of nodes at each step grows

exponentially with the number of steps that have been taken. Empirical results suggest that for blind search, this divide-and-conquer strategy is indeed effective. Unfortunately, other results, de Champeau and Sint suggest that for informed, heuristic search it is much less likely to be so. Figure 4.2 shows why bidirectional search may be ineffective. The two searches may pass each other, resulting in more work than it would have taken for one of them, on Its own, to have finished.

However, if individual forward and backward steps are performed as specified by a program that has been carefully constructed to exploit each in exactly those situations where it can be the most profitable, the results can be more encouraging. In fact, many successful AI applications have been written using a combination of forward and backward reasoning, and most AI programming environments provide explicit support for such hybrid reasoning.

Although in principle the same set of rules can be used for both forward and backward reasoning, in practice it has proved useful to define two classes of rules, each of which encodes a particular kind of knowledge.

- Forward rules, which encode knowledge about how to respond to certain input configurations.
- Backward rules, which encode knowledge about how to achieve particular goals.
- By separating rules into these two classes, we essentially add to each rule an additional piece of information, namely how it should be used in problem solving.

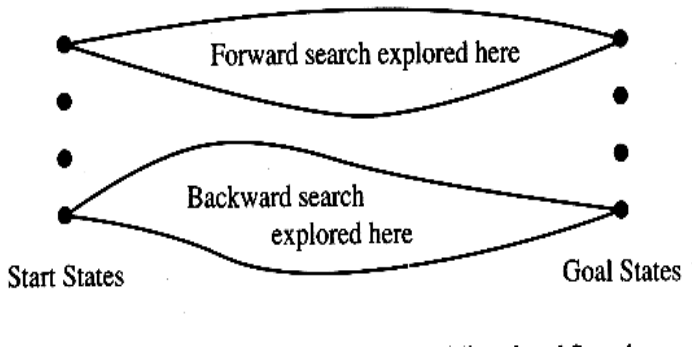


Figure 4.2: A Bad Use of Heuristic Bi-directional Search

4.3 Forward Chaining System

In a forward chaining system, the facts in the system are represented in a *working memory*, which is continually updated. Rules in the system represent possible actions to take when specified conditions hold on items in the working memory - they are sometimes called condition-action rules. The conditions are usually *patterns* that must *match* items in the working memory, while the actions usually involve *adding* or *deleting* items from the working memory. The interpreter controls the application of the rules, given the working memory, thus controlling the system's activity. It is based on a cycle of activity sometimes known as a *recognise-act* cycle. The system first checks to find all the rules whose conditions hold, given the current state of working memory. It then selects one and performs the actions in the action part of the rule. (The selection of a rule to fire is based on fixed strategies, known as *conflict resolution* strategies.) The actions will result in a new working memory, and the cycle begins again. This cycle will be repeated until either no rules fire, or some specified goal state is satisfied.

4.4 Backward Chaining System

If you do know what the conclusion might be, or have some specific hypothesis to test, forward chaining systems may be inefficient. You could keep on forward chaining until no more rules apply or you have added your hypothesis to the working memory. But in the process the system is likely to do a lot of irrelevant work, adding uninteresting conclusions to working memory. To avoid this we can use *backward chaining* systems.

Given a goal state to try and prove the system will first check to see if the goal matches the initial facts given. If it does, then that goal succeeds. If it doesn't the system will look for rules whose conclusions (previously referred to as *actions*) match the goal. One such rule will be chosen, and the system will then try to prove any facts in the preconditions of the rule using the same procedure, setting these as new goals to prove. Note that a backward chaining system does not need to update a working memory. Instead it needs to keep track of what goals it needs to prove to prove its main hypothesis.

4.5 Conflict Resolution

The result of the matching process is a list of rules whose antecedents have matched the current state description along with whatever variable bindings were generated by the matching process. It is the job of the search method to decide on the order in which rules will be applied. But sometimes it is useful to incorporate some of that decision making into the matching process. This phase of the matching process is then called conflict resolution.

There are three basic approaches to the problem of conflict resolution in a production system:

- Assign a preference based on the rule that matched.
- Assign a preference based on the objects that matched.
- Assign a preference based on the action that the matched rule would perform.

4.6 Use of No Backtrack

The real world is unpredictable, dynamic and uncertain. A robot cannot hope maintain a correct and complete description of the world. This means that robot does not consider the trade-off between devising and executing plans. This trade-off has several aspects. For one thing, robot may not possess enough information about the world for it to do any useful planning. In this case, it mostly first engages in information gathering activity. Furthermore, once it begins executing a plan, the robot most continually monitors the results of its actions. If the result is unexpected, then re-planning may be necessary.

Since robots operate in the real world, so searching and backtracking is a costly affair. Consider an example of an AI-first search for moving furniture into a room, operating in a simulated world with full information. Preconditions of operators can be checked quickly, and if an operator fails to apply, another can be tried checking preconditions in the real world, however, can be time consuming if the robot does not have full information. These problems can be solved by the adopting the approach of non-back.

4.7 Summary

In this lesson we have seen how to represent knowledge declaratively in rule-based systems and how to reason with that knowledge. A declarative representation is one in

which knowledge is specified, but the use to which that knowledge is to be put is not given where as a procedural representation is one in which the control information that is necessary to use the knowledge is considered to be embedded in the knowledge itself.

In PROLOG and many theorem-proving systems, rules are indexed by the predicates they contain, so all the rules that could be applicable to proving a particular fact can be accessed fairly quickly. The method of reasoning backward from the desired final state is called goal-directed reasoning.

Backward-chaining systems, of which PROLOG is an example, are good for goal directed problem solving. Backward-chaining systems usually use depth-first backtracking to select individual rules, but forward-chaining systems generally employ sophisticated conflict resolution strategies to choose among the applicable rules.

4.8 Keywords

Forward Reasoning, Conflict Resolution, Backward Reasoning, Forward Chaining System, Backward Chaining System, & Use of No Backtrack.

4.9 Check Your Progress

Q1. Differentiate between Rule-based architecture and non-production system architecture.

Q2. What do you understand by forward and backward reasoning?

Q3. Write short note on the following:

a. Conflict Resolution
b. Rule Based System
c. Set of Support Resolution Strategy
d. Use of No Backtrack

4.10 Reference/Suggested Reading

- Foundations of Artificial Intelligence and Expert System - V S Janakiraman, K Sarukesi, & P Gopalakrishanan, Macmillan Series.
- Artificial Intelligence – E. Rich and K. Knight

- Principles of Artificial Intelligence – Nilsson
- Expert Systems-Paul Harmon and David King, Wiley Press.
- Rule Based Expert System-Bruce G. Buchanan and Edward H. Shortliffe, eds., Addison Wesley.
- Introduction to Artificial Intelligence and Expert System- Dan W. Patterson, PHI, Feb., 2003.

SUBJECT: ARTIFICIAL INTELLIGENCE	
COURSE CODE: MCA-23	**AUTHOR:**
LESSON NO. 5	**VETTER:**
REASONING AND UNCERTAINTY	

UNIT STRUCTURE

5.0 Objectives

5.1 Symbolic reasoning under uncertainty

5.2 Non-monotonic reasoning

5.3 Uncertainty

5.4 Probability

5.5 Bayes' Theorem

5.6 Bayesian Networks

5.7 Rule-based Systems

5.8 Summary

5.9 Check Your Progress

5.10 Reference/Suggested Reading

5.0 Objective

In this lesson various structured knowledge techniques via semantic nets or networks, frames, scripts & conceptual dependency are discussed. It shows how knowledge is actually picturized and how effectively it resembles the representation of knowledge in human brain. After completion of this module, students come to know how to represent or handle problem(s) in AI.

5.1 Symbolic Reasoning Under Uncertainty

- The reasoning is the act of deriving a conclusion from certain properties using a given methodology.
- The reasoning is a process of thinking; reasoning is logically arguing; reasoning is drawing the inference.

- When a system is required to do something, that it has not been explicitly told how to do, it must reason. It must figure out what it needs to know from what it already knows.
- Many types of Reasoning have been identified and recognized, but many questions regarding their logical and computational properties still remain controversial.
- The popular methods of Reasoning include abduction, induction, model-based, explanation and confirmation. All of them are intimately related to problems of belief revision and theory development, knowledge absorption, discovery, and learning.

Logical Reasoning

- Logic is a language for reasoning. It is a collection of rules called Logic arguments, we use when doing logical reasoning.
- The logic reasoning is the process of drawing conclusions from premises using rules of inference.
- The study of logic divided into formal and informal logic. The formal logic is sometimes called symbolic logic.
- Symbolic logic is the study of symbolic abstractions (construct) that capture the formal features of logical inference by a formal system.
- The formal system consists of two components, a formal language plus a set of inference rules.
- The formal system has axioms. Axiom is a sentence that is always true within the system.
- Sentences derived using the system's axioms and rules of derivation called theorems.
- The Logical Reasoning is of our concern in AI.

Approaches to Reasoning

- There are three different approaches to reasoning under uncertainties.
  1. Symbolic reasoning
  2. Statistical reasoning
  3. Fuzzy logic reasoning Symbolic Reasoning

- The basis for intelligent mathematical software is the integration of the “power of symbolic mathematical tools” with the suitable “proof technology”.
- Mathematical reasoning enjoys a property called monotonicity, that says, “If a conclusion follows from given premises A, B, C… then it also follows from any larger set of premises, as long as the original premises A, B, C.. included.”
- Moreover, Human reasoning is not monotonic.
- People arrive at conclusions only tentatively; based on partial or incomplete information, reserve the right to retract those conclusions while they learn new facts. Such reasoning non-monotonic, precisely because the set of accepted conclusions have become smaller when the set of premises expanded.

Formal Logic

Moreover, The Formal logic is the study of inference with purely formal content, i.e. where content made explicit.

Examples – Propositional logic and Predicate logic.

- Here the logical arguments are a set of rules for manipulating symbols. The rules are of two types,
  1. Syntax rules: say how to build meaningful expressions.
  2. Inference rules: say how to obtain true formulas from other true formulas.
- Moreover, Logic also needs semantics, which says how to assign meaning to expressions. Uncertainty in Reasoning
- The world is an uncertain place; often the Knowledge is imperfect which causes uncertainty.
- So, therefore reasoning must be able to operate under uncertainty.
- Also, AI systems must have the ability to reason under conditions of uncertainty. Monotonic Reasoning
- A reasoning process that moves in one direction only.
- Moreover, the number of facts in the knowledge base is always increasing.
- The conclusions derived are valid deductions and they remain so. A monotonic logic cannot handle

1. Reasoning by default: because consequences may derive only because of lack of evidence to the contrary.

2. Abductive reasoning: because consequences only deduced as most likely explanations.
3. Belief revision: because new knowledge may contradict old beliefs.

5.2 Non-Monotonic Reasoning

The definite clause logic is **monotonic** in the sense that anything that could be concluded before a clause is added can still be concluded after it is added; adding knowledge does not reduce the set of propositions that can be derived.

A logic is **non-monotonic** if some conclusions can be invalidated by adding more knowledge. The logic of definite clauses with negation as failure is non-monotonic. Non-monotonic reasoning is useful for representing defaults. A **default** is a rule that can be used unless it overridden by an exception.

For example, to say that *b* is normally true if *c* is true, a knowledge base designer can write a rule of the form

$b \leftarrow c \wedge \sim ab_a$.

where ab_a is an atom that means abnormal with respect to some aspect *a*. Given *c*, the agent can infer *b* unless it is told ab_a. Adding ab_a to the knowledge base can prevent the conclusion of *b*.

Rules that imply ab_acan be used to prevent the default under the conditions of the body of the rule.

Example 5.27: Suppose the purchasing agent is investigating purchasing holidays. A resort may be adjacent to a beach or away from a beach. This is not symmetric; if the resort was adjacent to a beach, the knowledge provider would specify this. Thus, it is reasonable to have the clause

$away\_from\_beach \leftarrow \sim on\_beach$.

This clause enables an agent to infer that a resort is away from the beach if the agent is not told it is adjacent to a beach.

A **cooperative system** tries to not mislead. If we are told the resort is on the beach, we would expect that resort users would have access to the beach. If they have access to a beach, we would expect them to be able to swim at the beach. Thus, we would expect the following defaults: $beach\_access \leftarrow on\_beach \wedge \sim ab_{beach\_access}$.

$swim\_at\_beach \leftarrow beach\_access \wedge \sim ab_{swim\_at\_beach}$.

A cooperative system would tell us if a resort on the beach has no beach access or if there is no swimming. We could also specify that, if there is an enclosed bay and a big city, then there is no swimming, by default:

$ab_{swim\_at\_beach} \leftarrow enclosed\_bay \wedge big\_city \wedge \sim ab_{no\_swimming\_near\_city}$.

We could say that British Columbia is abnormal with respect to swimming near cities:

$ab_{no\_swimming\_near\_city} \leftarrow in\_BC \wedge \sim ab_{BC\_beaches}$.

Given only the preceding rules, an agent infers *away_from_beach*. If it is then told *on_beach*, it can no longer infer *away_from_beach*, but it can now infer *beach_access* and *swim_at_beach*. If it is also told *enclosed_bay* and *big_city*, it can no longer infer *swim_at_beach*. However, if it is then told *in_BC*, it can then infer *swim_at_beach*.

By having defaults of what is normal, a user can interact with the system by telling it what is abnormal, which allows for economy in communication. The user does not have to state the obvious.

One way to think about non-monotonic reasoning is in terms of **arguments**. The rules can be used as components of arguments, in which the negated abnormality gives a way to undermine arguments. Note that, in the language presented, only positive arguments exist that can be undermined. In more general theories, there can be positive and negative arguments that attack each other.

5.3 Uncertainty

Till now, we have learned knowledge representation using first-order logic and propositional logic with certainty, which means we were sure about the predicates. With this knowledge representation, we might write A→B, which means if A is true then B is true, but consider a situation where we are not sure about whether A is true or not then we cannot express this statement, this situation is called uncertainty.

So to represent uncertain knowledge, where we are not sure about the predicates, we need uncertain reasoning or probabilistic reasoning.

Causes of uncertainty:

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

1. Information occurred from unreliable sources.
2. Experimental Errors
3. Equipment fault

4. Temperature variation
5. Climate change.

Probabilistic reasoning

Probabilistic reasoning is a way of knowledge representation where we apply the concept of probability to indicate the uncertainty in knowledge. In probabilistic reasoning, we combine probability theory with logic to handle the uncertainty.

We use probability in probabilistic reasoning because it provides a way to handle the uncertainty that is the result of someone's laziness and ignorance.

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

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

As probabilistic reasoning uses probability and related terms, so before understanding probabilistic reasoning, let's understand some common terms:

5.4 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:

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

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:

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

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)**.

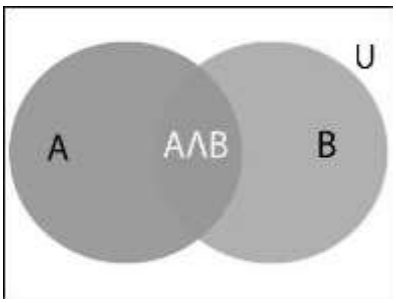


Figure 5.1 intersection of sets

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.

5.5 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.

It is a way to calculate the value of P(B|A) with the knowledge of P(A|B).

Bayes' theorem allows updating the probability prediction of an event by observing new information of the real world.

Example: If cancer corresponds to one's age then by using Bayes' theorem, we can determine the probability of cancer more accurately with the help of age.

Bayes' theorem can be derived using product rule and conditional probability of event A with known event B:

As from product rule we can write:

1. P(A ∧ B)= P(A|B) P(B) or

Similarly, the probability of event B with known event A:

1. P(A ∧ B)= P(B|A) P(A)

Equating right hand side of both the equations, we will get:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \qquad \text{……...(a)}$$

The above equation (a) is called as **Bayes' rule** or **Bayes' theorem**. This equation is basic of most modern AI systems for **probabilistic inference**. It shows the simple relationship between joint and conditional probabilities. Here, P(A|B) is known as **posterior**, which we need to calculate, and it will be read as Probability of hypothesis A when we have occurred an evidence B. P(B|A) is called the likelihood, in which we consider that hypothesis is true, then we calculate the probability of evidence.

P(A) is called the **prior probability**, probability of hypothesis before considering the evidence, P(B) is called **marginal probability**, pure probability of an evidence.

In the equation (a), in general, we can write P (B) = P(A)*P(B|Ai), hence the Bayes' rule can be written as:

$$P(A_i|B) = \frac{P(A_i) * P(B|A_i)}{\sum_{i=1}^{k} P(A_i) * P(B|A_i)}$$

Where A1, A2, A3,. , An is a set of mutually exclusive and exhaustive events.

Applying Bayes' rule:

Bayes' rule allows us to compute the single term P(B|A) in terms of P(A|B), P(*B*), and P(A). This is very useful in cases where we have a good probability of these three terms and want to determine the fourth one. Suppose we want to perceive the effect of some unknown cause, and want to compute that cause, then the Bayes' rule becomes:

$$\mathbf{P(cause|effect)} = \frac{\mathrm{P(effect|cause)\ P(cause)}}{\mathrm{P(effect)}}$$

Example-1:

Question: what is the probability that a patient has diseases meningitis with a stiff neck?

Given Data:

A doctor is aware that disease meningitis causes a patient to have a stiff neck, and it occurs 80% of the time. He is also aware of some more facts, which are given as follows:

- The Known probability that a patient has meningitis disease is 1/30,000.
- The Known probability that a patient has a stiff neck is 2%.

Let a be the proposition that patient has stiff neck and b be the proposition that patient has meningitis. , so we can calculate the following as:

P(a|b) = 0.8 P(b) = 1/30000 P(a)= .02

$$P(b|a) = \frac{P(a|b)P(b)}{P(a)} = \frac{0.8*(\frac{1}{30000})}{0.02} = 0.001333333.$$

Hence, we can assume that 1 patient out of 750 patients has meningitis disease with a stiff neck.

Example-2:

Question: From a standard deck of playing cards, a single card is drawn. The probability that the card is king is 4/52, then calculate posterior probability P(King|Face), which means the drawn face card is a king card.

$$P(king|face) = \frac{P(Face|king)*P(King)}{P(Face)} \quad .....(i)$$

Solution:

P(king): probability that the card is King= 4/52= 1/13 P(face): probability that a card is a face card= 3/13

P(Face|King): probability of face card when we assume it is a king = 1 Putting all

$$P(king|face) = \frac{1*(\frac{1}{13})}{(\frac{3}{13})} = 1/3,$$ **it is a probability that a face card is a king card.**

values in equation (i) we will get:

Application of Bayes' theorem in Artificial intelligence:

Following are some applications of Bayes' theorem:

- It is used to calculate the next step of the robot when the already executed step is given.
- Bayes' theorem is helpful in weather forecasting.
- It can solve the Monty Hall problem.

5.6 Bayesian Belief Network

Bayesian belief network is key computer technology for dealing with probabilistic events and to solve a problem which has uncertainty. We can define a Bayesian network as:

"A Bayesian network is a probabilistic graphical model which represents a set of variables and their conditional dependencies using a directed acyclic graph."

It is also called a **Bayes network, belief network, decision network**, or **Bayesian model**. Bayesian networks are probabilistic, because these networks are built from a **probability distribution**, and also use probability theory for prediction and anomaly detection.

Real world applications are probabilistic in nature, and to represent the relationship between multiple events, we need a Bayesian network. It can also be used in various tasks including **prediction, anomaly detection, diagnostics, automated insight, reasoning, time series prediction**, and **decision making under uncertainty**.

Bayesian Network can be used for building models from data and experts opinions, and it consists of two parts:

- **Directed Acyclic Graph**
- **Table of conditional probabilities.**

The generalized form of Bayesian network that represents and solve decision problems under uncertain knowledge is known as an **Influence diagram**.

A Bayesian network graph is made up of nodes and Arcs (directed links),

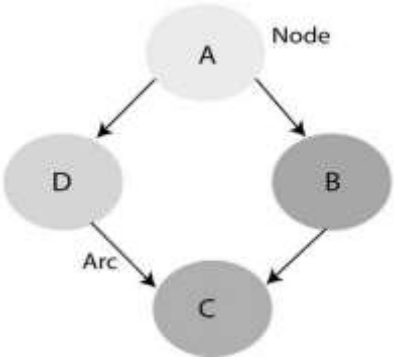


Figure 5.2 Bayesian Network

- Each node corresponds to the random variables, and a variable can be continuous or discrete.
- Arc or directed arrows represent the causal relationship or conditional probabilities between random variables. These directed links or arrows connect the pair of nodes in the graph. These links represent that one node directly influence the other node, and if there is no directed link that means that nodes are independent with each other
- In the above diagram, A, B, C, and D, are random variables represented by the nodes of the network graph.
- If we are considering node B, which is connected with node A by a directed arrow, then node A is called the parent of Node B.
- Node C is independent of node A.

The Bayesian network has mainly two components:

- Causal Component
- Actual numbers

Each node in the Bayesian network has condition probability distribution **P(Xi |Parent(Xi))**, which determines the effect of the parent on that node.

Bayesian network is based on Joint probability distribution and conditional probability. So let's first understand the joint probability distribution:

Joint probability distribution:

If we have variables x1, x2, x3,. , xn, then the probabilities of a different combination

of x1, x2, x3.. xn, are known as Joint probability distribution.

P[x1, x2, x3,....., xn], it can be written as the following way in terms of the joint probability distribution.

= P[x1| x2, x3,....., xn]P[x2, x3,. , xn]

= P[x1| x2, x3,....., xn]P[x2|x3,....., xn] P[xn-1|xn]P[xn].

In general for each variable Xi, we can write the equation as:

$P(X_i|X_{i-1},....., X1) = P(X_i |Parents(X_i))$

Explanation of Bayesian network:

Let's understand the Bayesian network through an example by creating a directed acyclic graph:

Example: Harry installed a new burglar alarm at his home to detect burglary. The alarm reliably responds at detecting a burglary but also responds for minor earthquakes. Harry has two neighbors David and Sophia, who have taken a responsibility to inform Harry at work when they hear the alarm. David always calls Harry when he hears the alarm, but sometimes he got confused with the phone ringing and calls at that time too. On the other hand, Sophia likes to listen to high music, so sometimes she misses to hear the alarm. Here we would like to compute the probability of Burglary Alarm.

Problem:

Calculate the probability that alarm has sounded, but there is neither a burglary, nor an earthquake occurred, and David and Sophia both called the Harry.

Solution:

- The Bayesian network for the above problem is given below. The network structure is showing that burglary and earthquake is the parent node of the alarm and directly affecting the probability of alarm's going off, but David and Sophia's calls depend on alarm probability.
- The network is representing that our assumptions do not directly perceive the burglary and also do not notice the minor earthquake, and they also not confer before calling.

- The conditional distributions for each node are given as conditional probabilities table or CPT.
- Each row in the CPT must be sum to 1 because all the entries in the table represent an exhaustive set of cases for the variable.
- In CPT, a boolean variable with k boolean parents contains 2^K probabilities. Hence, if there are two parents, then CPT will contain 4 probability values

List of all events occurring in this network:

- **Burglary (B)**
- **Earthquake(E)**
- **Alarm(A)**
- **David Calls(D)**
- **Sophia calls(S)**

We can write the events of problem statement in the form of probability: **P[D, S, A, B, E]**, can rewrite the above probability statement using joint probability distribution:

P[D, S, A, B, E]= P[D | S, A, B, E]. P[S, A, B, E]

=P[D | S, A, B, E]. P[S | A, B, E]. P[A, B, E]

= P [D| A]. P [S| A, B, E]. P[A, B, E]

= P[D | A]. P[S | A]. P[A| B, E]. P[B, E]

= P[D | A]. P[S | A]. P[A| B, E]. P[B |E]. P[E]

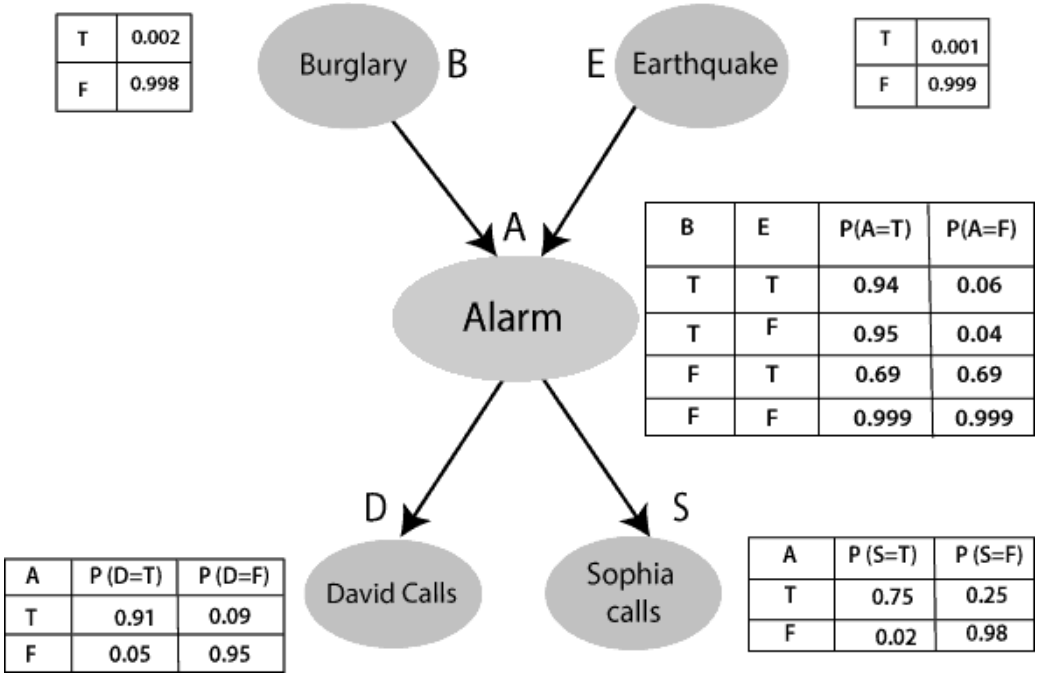


Figure 5.3 Probability of events

Let's take the observed probability for the Burglary and earthquake component: P (B= True) = 0.002, which is the probability of burglary.

P (B= False) = 0.998, which is the probability of no burglary.

P (E= True) = 0.001, which is the probability of a minor earthquake

P (E= False)= 0.999, Which is the probability that an earthquake not occurred. We can provide the conditional probabilities as per the below tables: **Conditional probability table for Alarm A:**

The Conditional probability of Alarm A depends on Burglar and earthquake: The Conditional probability of Alarm A depends on Burglar and earthquake:

B	***E***	***P(A= True)***	***P(A= False)***
True	True	0.94	0.06
True	False	0.95	0.04
False	True	0.31	0.69
False	False	0.001	0.999

Conditional probability table for David Calls:

The Conditional probability of David that he will call depends on the probability of Alarm.

A	*P(D= True)*	*P(D= False)*
True	0.91	0.09
False	0.05	0.95

Conditional probability table for Sophia Calls:

The Conditional probability of Sophia that she calls is depending on its Parent Node "Alarm."

A	*P(S= True)*	*P(S= False)*
True	0.75	0.25
False	0.02	0.98

From the formula of joint distribution, we can write the problem statement in the form of probability distribution:

P(S, D, A, ¬B, ¬E) = P (S|A) *P (D|A)*P (A|¬B ^ ¬E) *P (¬B) *P (¬E).

= 0.75* 0.91* 0.001* 0.998*0.999

= 0.00068045.

Hence, a Bayesian network can answer any query about the domain by using Joint distribution.

The semantics of Bayesian Network:

There are two ways to understand the semantics of the Bayesian network, which is given below:

1. **To understand the network as the representation of the Joint probability distribution.**

It is helpful to understand how to construct the network.

2. **To understand the network as an encoding of a collection of conditional independence statements.**

It is helpful in designing inference procedure.

5.6 Summary

In this lesson we have investigated different types of structural knowledge representation methods. We considered associative networks (semantic net), a

representation based on a structure of linked nodes (concepts) and arcs (relations) connecting the nodes. With these networks we saw how related concepts could be structured into cohesive units and exhibited as graphical representation. A frame is a collection of attributes (usually called slots) and associated values (and possibly constraints on values) that describe some entity in the world. In this lesion we also described a special frame-like structure called scripts. Scripts are used to represent stereotypical patterns for commonly occurring events. Like a play scripts contains actors, roles, props, and scenes, which combine to represent a familiar situation. Scripts have been used in a number of programs, which read and "understood" language in the form of stories.

5.7 Key Words

Semantic Net Slots, Slots, Frame, Scripts & Exceptions & Defaults

5.8 Check Your Progress

Answer the following questions

Q1. Explain & distinguish between the following: -

a. Associative Network Structure

b. Frame Structure

Q2. What are the main difference between scripts and frame structure?

Q3. Write short note on the following: -

a. Exception & Defaults

b. Semantic Net

c. Slots

5.9 Reference/Suggested Reading

- Foundations of Artificial Intelligence and Expert System - V S Janakiraman, K Sarukesi, & P Gopalakrishanan, Macmillan Series.
- Artificial Intelligence – E. Rich and K. Knight
- Principles of Artificial Intelligence – Nilsson

- Expert Systems-Paul Harmon and David King, Wiley Press.
- Rule Based Expert System-Bruce G. Buchanan and Edward H. Shortliffe, eds., Addison Wesley.
- Introduction to Artificial Intelligence and Expert System- Dan W. Patterson, PHI, Feb., 2003.

SUBJECT: ARTIFICIAL INTELLIGENCE	
COURSE CODE: MCA-23	**AUTHOR:**
LESSON NO. 6	**VETTER:**
FUZZY LOGIC	

UNIT STRUCTURE

6.0 Objectives

6.1 Use of Certainty Factors

6.3 Fuzzy Logic

6.4 Concept of Learning

6.5 Learning Automation

6.6 Genetic Algorithm

6.7 Learning by Induction

6.8 Neural Networks

6.9 Summary

6.10 Check Your Progress

6.11 Reference/Suggested Reading

6.0 OBJECTIVE

Learning is a continues process of knowledge refinement. This lesson discuss about various learning techniques, Probabilistic Reasoning, Use of Certainty Factors, Fuzzy Logic, Concept of Learning, Learning Automation, Genetic Algorithm, Learning by Induction and Neural Networks. Upon completion of this lesson students come to know how a machine acquire knowledge and better understanding of the terms like genetic algorithm and neural networks.

6.1 USE OF CERTAINTY FACTORS

MYCIN uses measures of both belief and disbelief to represent degrees of confirmation and disconfirmation respectively in a given hypothesis. The basic measure of belief, denoted by $MB(H,E)$, is actually a measure of increased belief in hypothesis H due to the evidence E. This is roughly equivalent to the estimated increase in probability of $P(H/E)$ over $P(H)$ given by an expert as a result of the knowledge gained by E. A value of 0 corresponds to no increase in belief and 1 corresponds to maximum increase or absolute belief. Likewise, $MD(H,E)$ is a measure of the increased disbelief in hypothesis H due to evidence E. MD ranges from 0 to +1 with +1 representing maximum increase in disbelief, (total disbelief) and 0 representing no increase. In both measures, the evidence E may be absent or may be replaced with another hypothesis, $MB(H_1, H_2)$. This represents the increased belief in H_1 given H_2 is true.

In an attempt to formalize the uncertainty measure in MYCIN, definitions of MB and MD have been given in terms of prior and conditional probabilities. It should be remembered, however, the actual values are often subjective probability estimates provided by a physician. We have for the definitions.

$$MB(H,E) = \begin{cases} 1 & \text{If } P(H) = 1 \\ \dfrac{\max[P(H|E), P(H)] - P(H)}{1 - P(H)} & \text{otherwise} \end{cases}$$

(6.11)

$$MB(H,E) = \begin{cases} 1 & \text{If } P(H) = 1 \\ \dfrac{\min[P(H|E), P(H)] - P(H)}{0 = P(H)} & \text{otherwise} \end{cases}$$

(6.12)

Note that when $0 < P(H) < 1$, and E and H are independent (So $P(H|E) = P(H)$, then MB = MD = 0. This would be the case if E provided no useful information.

The two measures MB and MD are combined into a single measure called the certainty factor (CF), defined by

$$CF(H,E) = MB(H,E) - MD(H,E) \tag{6.13}$$

Note that the value of CF ranges from –1 (certain disbelief) to +1 (certain belief). Furthermore, a value of CF = 0 will result if E neither confirms nor unconfirms H (E and H are independent).

6.3 FUZZY LOGIC

Fuzzy logic has rapidly become one of the most successful of today's technologies for developing sophisticated control systems. The reason for which is very simple. Fuzzy logic addresses such applications perfectly as it resembles human decision making with an ability to generate precise solutions from certain or approximate information. It fills an important gap in engineering design methods left vacant by purely mathematical approaches (e.g. linear control design), and purely logic-based approaches (e.g. expert systems) in system design.

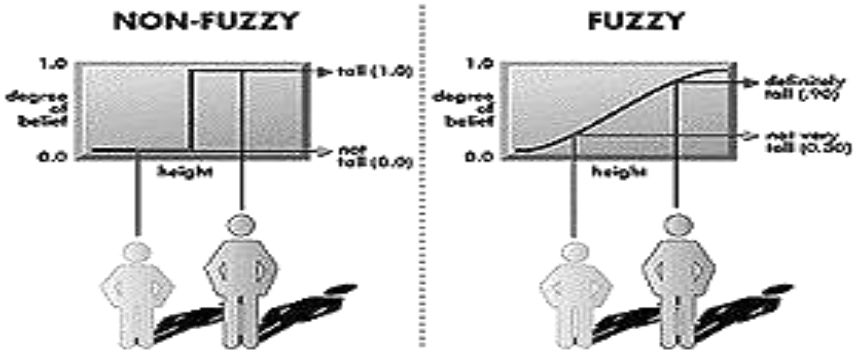


Figure 6.1 Non-fuzzy and fuzzy set

While other approaches require accurate equations to model real-world behaviors, fuzzy design can accommodate the ambiguities of real-world human language and logic. It provides both an intuitive method for describing systems in human terms and automates the conversion of those system specifications into effective models.

What does it offer?

The first applications of fuzzy theory were primaly industrial, such as process control

for cement kilns. However, as the technology was further embraced, fuzzy logic was used in more useful applications. In 1987, the first fuzzy logic-controlled subway was opened in Sendai in northern Japan. Here, fuzzy-logic controllers make subway journeys more comfortable with smooth braking and acceleration. Best of all, all the driver has to do is push the start button! Fuzzy logic was also put to work in elevators to reduce waiting time. Since then, the applications of Fuzzy Logic technology have virtually exploded, affecting things we use everyday.

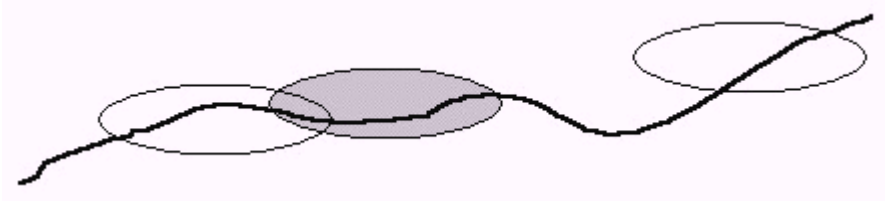
Take for example, the *fuzzy washing machine.* A load of clothes in it and press start, and the machine begins to churn, automatically choosing the best cycle. The fuzzy microwave, Place chili, potatoes, or etc in a *fuzzy microwave* and push single button, and it cooks for the right time at the proper temperature. The *fuzzy car*, manuvers itself by following simple verbal instructions from its driver. It can even stop itself when there is an obstacle immediately ahead using sensors. But, practically the most exciting thing about it, is the simplicity involved in operating it.

Fuzzy Rules

Human beings make decisions based on rules. Although, we may not be aware of it, all the decisions we make are all based on computer like if-then statements. If the weather is fine, then we may decide to go out. If the forecast says the weather will be bad today, but fine tomorrow, then we make a decision not to go today, and postpone it till tomorrow. Rules associate ideas and relate one event to another.

Fuzzy machines, which always tend to mimic the behavior of man, work the same way. However, the decision and the means of choosing that decision are replaced by fuzzy sets and the rules are replaced by fuzzy rules. Fuzzy rules also operate using a series of if-then statements. For instance, if X then A, if y then b, where A and B are all sets of X and Y. Fuzzy rules define fuzzy *patches*, which is the key idea in fuzzy logic.

A machine is made smarter using a concept designed by Bart Kosko called the Fuzzy Approximation Theorem(FAT). The FAT theorem generally states a finite number of patches can cover a curve as seen in the figure below. If the patches are large, then the rules are sloppy. If the patches are small then the rules are fine.



Fuzzy Patches

In a fuzzy system this simply means that all our rules can be seen as patches and the input and output of the machine can be associated together using these patches. Graphically, if the *rule patches* shrink, our fuzzy subset triangles gets narrower. Simple enough? Yes, because even novices can build control systems that beat the best math models of control theory. Naturally, it is *math-free* system.

Fuzzy Control

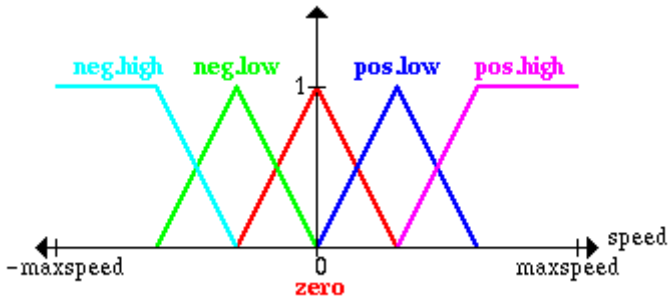
Fuzzy control, which directly uses fuzzy rules is the most important application in fuzzy theory. Using a procedure originated by Ebrahim Mamdani in the late 70s, three steps are taken to create a fuzzy controlled machine:

1) Fuzzification (Using membership functions to graphically describe a situation)
2) Rule evaluation (Application of fuzzy rules)
3) Defuzzification (Obtaining the crisp or actual results)

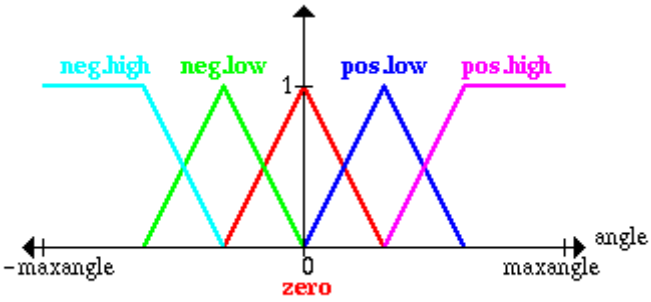
As a simple example on how fuzzy controls are constructed, consider the following classic situation: the inverted pendulum. Here, the problem is to balance a pole on a mobile platform that can move in only two directions, to the left or to the right. The angle between the platform and the pendulum and the angular velocity of this angle are chosen as the inputs of the system. The speed of the platform hence, is chosen as the corresponding output.

Step 1

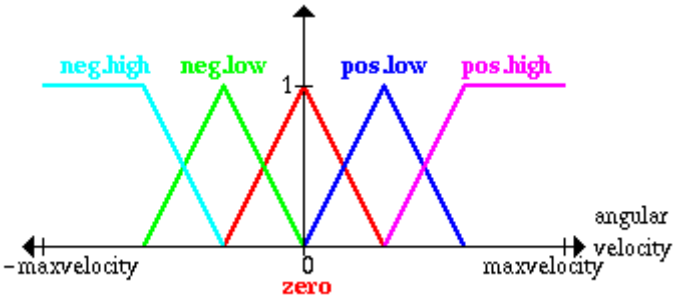
First of all, the different levels of output (high speed, low speed etc.) of the platform is defined by specifying the membership functions for the fuzzy_sets. The graph of the function is shown below



Similary, the different angles between the platform and the pendulum and...



the angular velocities of specific angles are also defined



Note: For simplicity, it is assumed that all membership functions are spreaded equally. Hence, this explains why no actual scale is included in the graphs.

Step 2

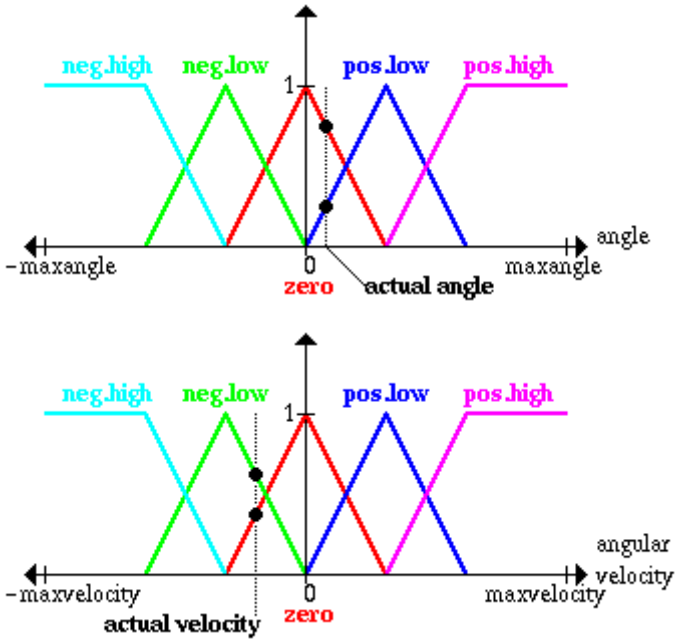
The next step is to define the fuzzy rules. The fuzzy rules are merely a series of if-then statements as mentioned above. These statements are usually derived by an expert to achieve optimum results. Some examples of these rules are:

i) If angle is zero and angular velocity is zero then speed is also zero.

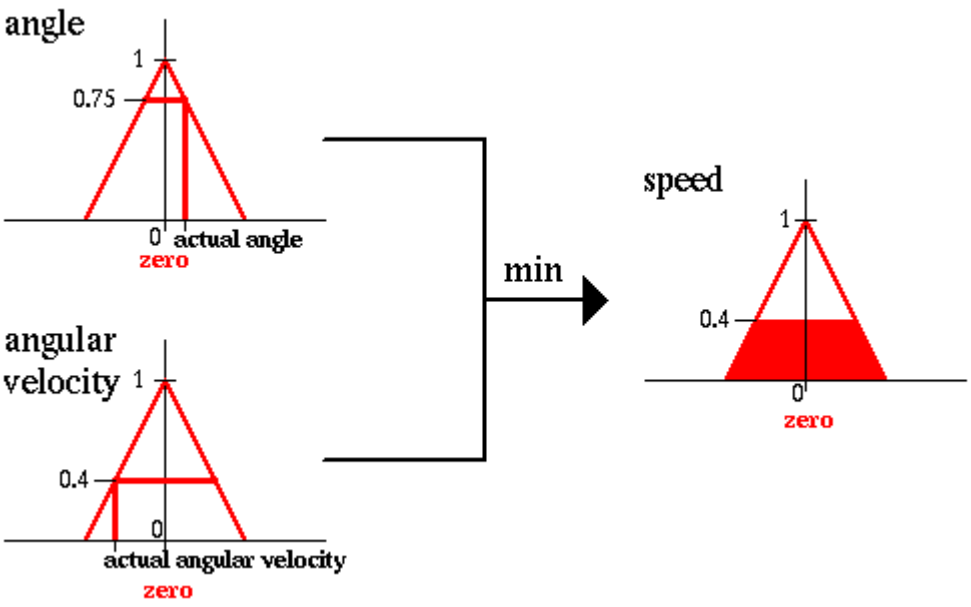
ii) If angle is zero and angular velocity is low then the speed shall be low.

The full set of rules is summarized in the table below. The dashes are for conditions, which have no rules associated with them. This is don eto simplify the situation.

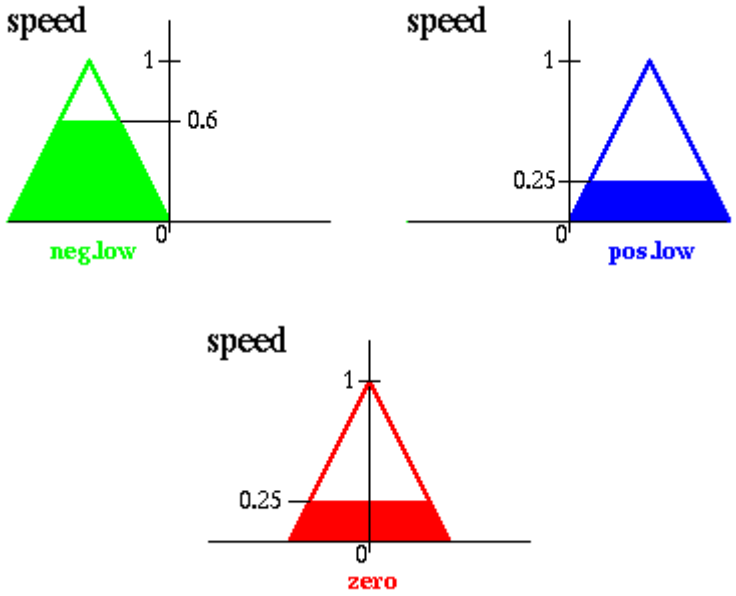
An application of these rules is shown using specific values for angle and angular velocities. The values used for this example are 0.75 and 0.25 for zero and positive-low angles, and 0.4 and 0.6 for zero and negative-low angular velocities. These points sre on the graphs below.



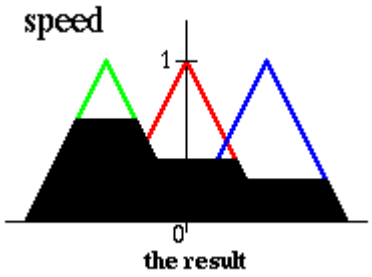
Consider the rule "if angle is zero and angular velocity is zero, the speed is zero". The actual value belongs to the fuzzy set zero to a degree of 0.75 for "angle" and 0.4 for "angular velocity". Since this is an AND operation, the minimum criterion is used , and the fuzzy set zero of the variable "speed" is cut at 0.4 and the patches are shaded up to that area. This is illustrated in the figure below.



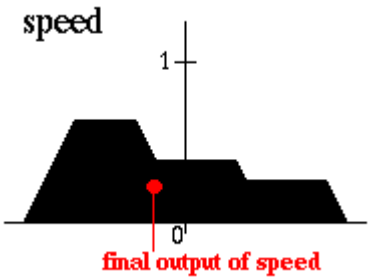
Similarly, the minimum criterion is used for the other three rule. The following figures show the result *patches* yielded by the rule "if angle is zero and angular velocity is negative low, the speed is negative low", "if angle is positive low and angular velocity is zero, then speed is positive low" and "if angle is positive low and angular velocity is negative low, the speed is zero".



The four results overlaps and is reduced to the following figure



Step 3: The result of the fuzzy controller as of know is a fuzzy set (of speed). In order to choose an appropriate representative value as the final output(crisp values), defuzzification must be done. There are numerous defuzzification methods, but the most common one used is the center of gravity of the set as shown below.



What do you mean *fuzzy* ??!!

Before illustrating the mechanisms which make fuzzy logic machines work, it is important to realize what fuzzy logic actually is. Fuzzy logic is a superset of conventional (Boolean) logic that has been extended to handle the concept of partial truth- truth-values between "completely true" and "completely false". As its name suggests, it is the logic underlying modes of reasoning which are approximate rather than exact. The importance of fuzzy logic derives from the fact that most modes of human reasoning and especially common sense reasoning are approximate in nature.

The essential characteristics of fuzzy logic as founded by Zader Lotfi are as follows.

- In fuzzy logic, exact reasoning is viewed as a limiting case of approximate reasoning.
- In fuzzy logic everything is a matter of degree.
- Any logical system can be fuzzified.
- In fuzzy logic, knowledge is interpreted as a collection of elastic or, equivalently, fuzzy constraint on a collection of variables
- Inference is viewed as a process of propagation of elastic constraints.

The third statement hence, defines Boolean logic as a subset of Fuzzy logic.

Fuzzy Sets

Fuzzy Set Theory was formalized by Professor Lofti Zadeh at the University of California in 1965. What Zadeh proposed is very much a paradigm shift that first gained acceptance in the Far East and its successful application has ensured its adoption around the world.

A paradigm is a set of rules and regulations, which defines boundaries and tells us what to do to be successful in solving problems within these boundaries. For example the use of transistors instead of vacuum tubes is a paradigm shift - likewise the development of Fuzzy Set Theory from conventional bivalent set theory is a paradigm shift.

Bivalent Set Theory can be somewhat limiting if we wish to describe a 'humanistic' problem mathematically. For example, Fig 1 below illustrates bivalent sets to characterise the temperature of a room.

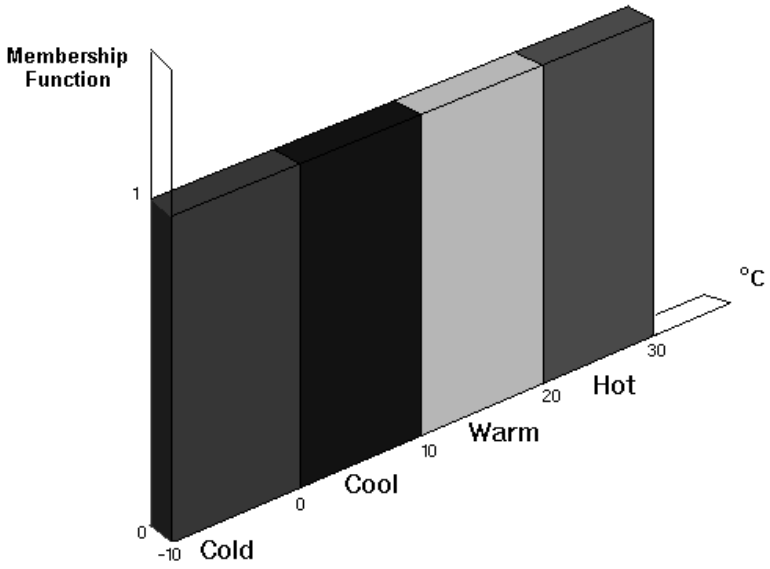


Figure 6.2 Bivalent sets to Characterize the Temp. of a room

The most obvious limiting feature of bivalent sets that can be seen clearly from the diagram is that they are mutually exclusive - it is not possible to have membership of more than one set (opinion would widely vary as to whether 50 degrees Fahrenheit is 'cold' or 'cool' hence the expert knowledge we need to define our system is mathematically at odds with the humanistic world). Clearly, it is not accurate to define a transiton from a quantity such as 'warm' to 'hot' by the application of one

degree Fahrenheit of heat. In the real world a smooth (unnoticeable) drift from warm to hot would occur.

This natural phenomenon can be described more accurately by Fuzzy Set Theory. Fig.2 below shows how fuzzy sets quantifying the same information can describe this natural drift.

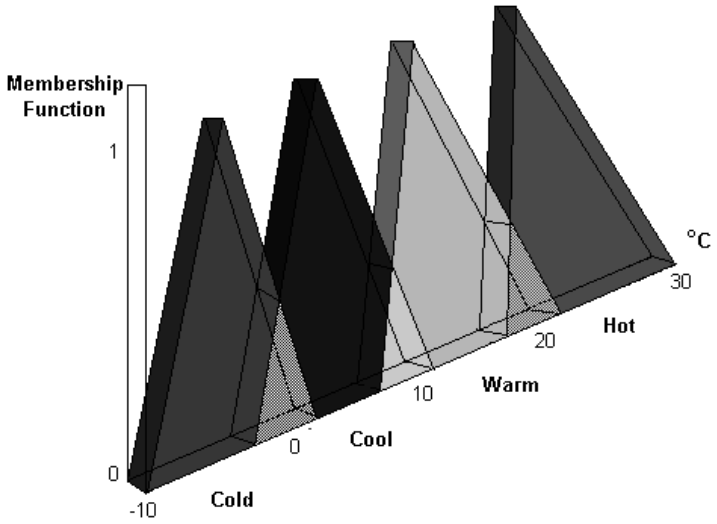


Figure 6.3 Fuzzy sets to characterize the Temp. of a room.

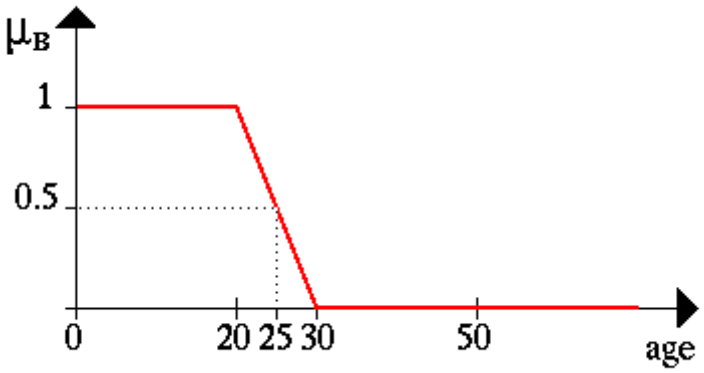
The whole concept can be illustrated with this example. Let's talk about people and "youthness". In this case the set S (the universe of discourse) is the set of people. A fuzzy subset YOUNG is also defined, which answers the question "to what degree is person x young?" To each person in the universe of discourse, we have to assign a degree of membership in the fuzzy subset YOUNG. The easiest way to do this is with a membership function based on the person's age.

young(x) = { 1, if age(x) <= 20,

(30-age(x))/10, if 20 < age(x) <= 30,

0, if age(x) > 30 }

A graph of this looks like:



Given this definition, here are some example values:

Person Age degree of youth

Johan 10 1.00

Edwin 21 0.90

Parthiban 25 0.50

Arosha 26 0.40

Chin Wei 28 0.20

Rajkumar 83 0.00

So given this definition, we'd say that the degree of truth of the statement "Parthiban is YOUNG" is 0.50.

Note: Membership functions almost never have as simple a shape as age(x). They will at least tend to be triangles pointing up, and they can be much more complex than that. Furthermore, membership functions so far is discussed as if they always are based on a single criterion, but this isn't always the case, although it is the most common case. One could, for example, want to have the membership function for YOUNG depend on both a person's age and their height (Arosha's short for his age). This is perfectly legitimate, and occasionally used in practice. It's referred to as a two-dimensional membership function. It's also possible to have even more criteria, or to

have the membership function depend on elements from two completely different universes of discourse.

Fuzzy Set Operations.

Union

The membership function of the Union of two fuzzy sets A and B with membership functions μ_A and μ_B respectively is defined as the maximum of the two individual membership functions. This is called the *maximum* criterion.

$$\mu_{A \cup B} = \max(\mu_A, \mu_B)$$

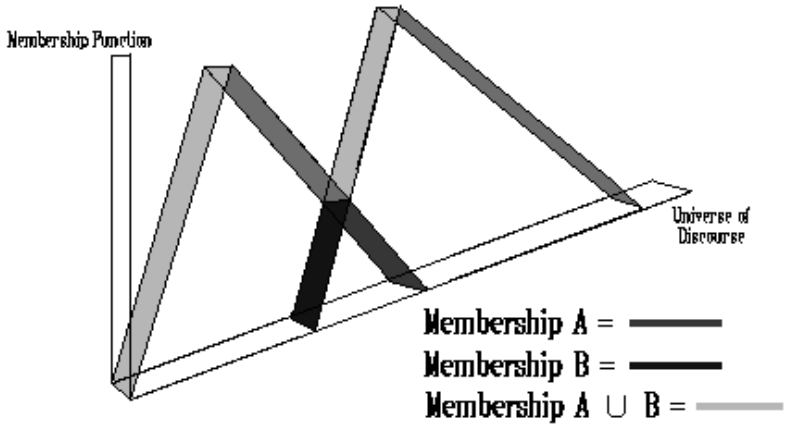


Figure 6.4 Union

The Union operation in Fuzzy set theory is the equivalent of the **OR** operation in Boolean algebra.

Intersection

The membership function of the Intersection of two fuzzy sets A and B with membership functions μ_A and μ_B respectively is defined as the minimum of the two individual membership functions. This is called the *minimum* criterion.

$$\mu_{A \cap B} = \min(\mu_A, \mu_B)$$

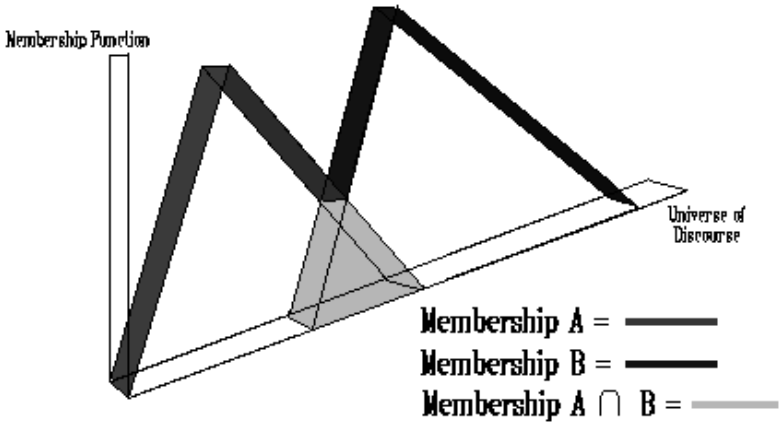


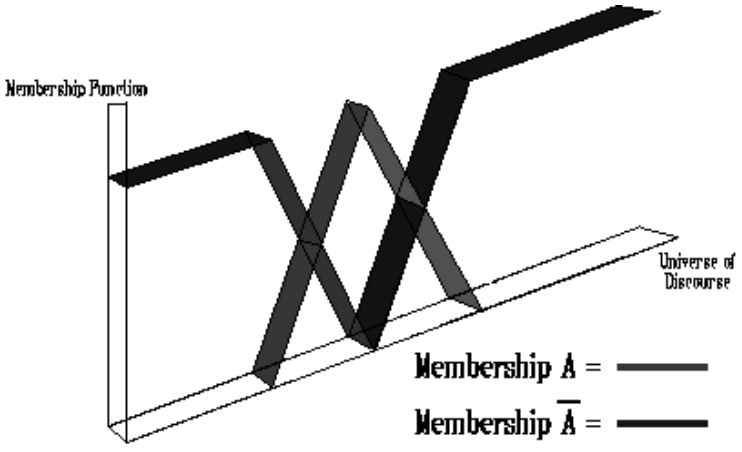
Figure 6.5 Intersection

The Intersection operation in Fuzzy set theory is the equivalent of the **AND** operation in Boolean algebra.

Complement

The membership function of the Complement of a Fuzzy set A with membership function μ_A is defined as the negation of the specified membership function. This is caleed the *negation* criterion.

$$\mu_{\overline{A}} = 1 - \mu_A$$



The Complement operation in Fuzzy set theory is the equivalent of the **NOT** operation in Boolean algebra.

The following rules which are common in classical set theory also apply to Fuzzy set theory.

De Morgans law

$$\overline{(A \cap B)} = \overline{A} \cap \overline{B}\ ,\ \overline{(A \cup B)} = \overline{A} \cap \overline{B}$$

Associativity

$$(A \cap B) \cap C = A \cap (B \cap C)$$

$$(A \cup B) \cup C = A \cup (B \cup C)$$

Commutativity

$$A \cap B = B \cap A\ ,\ A \cup B = B \cup A$$

Distributive

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$

Universe of Discourse

The Universe of Discourse is the range of all possible values for an input to a fuzzy system.

Fuzzy Set

A Fuzzy Set is any set that allows its members to have different grades of membership (membership function) in the interval [0,1].

Support

The Support of a fuzzy set F is the crisp set of all points in the Universe of Discourse U such that the membership function of F is non-zero.

Crossover point

The Crossover point of a fuzzy set is the element in U at which its membership function is 0.5.

Fuzzy Singleton

A Fuzzy singleton is a fuzzy set whose support is a single point in U with a membership function of one.

6.6 Genetic Algorithm

Genetic Algorithms allow you to explore a space of parameters to find solutions that score well according to a "fitness function". They are a way to implement *function optimization*: given a function g(x) (where x is typically a vector of parameter values), find the value of x that maximizes (or minimizes) g(x). This is an *unsupervised learning* problem—the right answer is not known beforehand. For pathfinding, given a starting position and a goal, x is the path between the two and g(x) is the cost of that path. Simple optimization approaches like hill-climbing will change x in ways that increase g(x). Unfortunately, in some problems, you reach "local maxima", values of x for which no nearby x has a greater value of g, but some faraway value of x is better. Genetic algorithms improve upon hill climbing by maintaining multiple x, and using evolution-inspired approaches like mutation and crossover to alter x. Both hill-climbing and genetic algorithms can be used to learn the best value of x. For path finding, however, we already have an algorithm (A*) to find the best x, so function optimization approaches are not needed.

Genetic Programming takes genetic algorithms a step further, and treats *programs* as the parameters. For example, you would be breeding path finding *algorithms* instead of *paths*, and your fitness function would rate each algorithm based on how well it does. For path finding, we already have a good algorithm and we do not need to evolve a new one.

It may be that as with neural networks, genetic algorithms can be applied to some portion of the path-finding problem. However, I do not know of any uses in this context. Instead, a more promising approach seems to be to use path finding, for which solutions are known, as one of many tools available to evolving agents.

6.8 Neural Networks

Neural networks are structures that can be "trained" to recognize patterns in inputs. They are a way to implement function approximation: given $y_1 = f(x_1)$, $y_2 = f(x_2)$, ..., $y_n = f(x_n)$, construct a function f' that approximates f. The approximate function f' is typically *smooth*: for x' close to x, we will expect that f'(x') is close to f'(x). Function approximation serves two purposes:

- **Size:** the representation of the approximate function can be significantly smaller than the true function.
- **Generalization:** the approximate function can be used on inputs for which we do not know the value of the function.

Neural networks typically take a vector of input values and produce a vector of output values. Inside, they train weights of "neurons". Neural networks use *supervised learning*, in which inputs and outputs are known and the goal is to build a representation of a function that will approximate the input to output mapping.

In path finding, the function is f(start, goal) = path. We do not already know the output paths. We could compute them in some way, perhaps by using A*. But if we are able to compute a path given (start, goal), then we already know the function f, so why bother approximating it? There is no use in generalizing f because we know it completely. The only potential benefit would be in reducing the size of the representation of f. The representation of f is a fairly simple algorithm, which takes little space, so I don't think that's useful either. In addition, neural networks produce a fixed-size output, whereas paths are variable sized.

Instead, function approximation may be useful to construct components of path finding. It may be that the movement cost function is unknown. For example, the cost of moving across an orc-filled forest may not be known without actually performing the movement and fighting the battles. Using function approximation, each time the forest is crossed, the movement cost f(number of orcs, size of forest) could be measured and fed into the neural network. For future pathfinding sessions, the new movement costs could be used to find better paths. Even when the function is unknown, function approximation is useful primarily when the function varies from

game to game. If a single movement cost applies every time someone plays the game, the game developer can precompute it beforehand.

Another function that is could benefit from approximation is the heuristic. The heuristic function in A* should estimate the minimum cost of reaching the destination. If a unit is moving along path P = p_1, p_2, ..., p_n, then after the path is traversed, we can feed n updates, g(p_i, p_n) = (actual cost of moving from i to n), to the approximation function h. As the heuristic gets better, A* will be able to run quicker.

Neural networks, although not useful for path finding itself, can be used for the functions used by A*. Both movement and the heuristic are functions that can be measured and therefore fed back into the function approximation.

The Backpropagation Algorithm

1. Propagates inputs forward in the usual way, i.e.

    - All outputs are computed using sigmoid thresholding of the inner product of the corresponding weight and input vectors.
    - All outputs at stage *n* are connected to all the inputs at stage *n*+1

2. Propagates the errors backwards by apportioning them to each unit according to the amount of this error the unit is responsible for.

We now derive the stochastic Backpropagation algorithm for the general case. The derivation is simple, but unfortunately the bookkeeping is a little messy.

- $\vec{x_j} =$ input vector for unit *j* (x_{ji} = *i*th input to the *j*th unit)
- $\vec{w_j} =$ weight vector for unit *j* (w_{ji} = weight on x_{ji})
- $z_j = \vec{w_j} \cdot \vec{x_j}$, the weighted sum of inputs for unit *j*
- $o_j = \sigma(z_j)$
- o_j = output of unit *j* ( )
- t_j = target for unit *j*

- *Downstream*(*j*) = set of units whose immediate inputs include the output of *j*
- *Outputs* = set of output units in the final layer

Since we update after each training example, we can simplify the notation somewhat by imagining that the training set consists of exactly one example and so the error can simply be denoted by *E*.

We want to calculate $\frac{\partial E}{\partial w_{ji}}$ for each input weight w_{ji} for each output unit *j*. Note first that since z_j is a function of w_{ji} regardless of where in the network unit *j* is located,

$$\begin{aligned}\frac{\partial E}{\partial w_{ji}} &= \frac{\partial E}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ji}} \\ &= \frac{\partial E}{\partial z_j} x_{ji}\end{aligned}$$

Furthermore, $\frac{\partial E}{\partial z_j}$ is the same regardless of which input weight of unit *j* we are trying to update. So we denote this quantity by δ_j.

Consider the case when $j \in Outputs$. We know

$$E = \frac{1}{2} \sum_{k \in Outputs} (t_k - \sigma(z_k))^2$$

Since the outputs of all units $k \neq j$ are independent of w_{ji}, we can drop the summation and consider just the contribution to *E* by *j*.

$$\begin{aligned}\delta_j = \frac{\partial E}{\partial z_j} &= \frac{\partial}{\partial z_j}\frac{1}{2}(t_j - o_j))^2 \\ &= -(t_j - o_j)\frac{\partial o_j}{\partial z_j} \\ &= -(t_j - o_j)\frac{\partial}{\partial z_j}\sigma(z_j) \\ &= -(t_j - o_j)(1 - \sigma(z_j))\sigma(z_j) \\ &= -(t_j - o_j)(1 - o_j)o_j\end{aligned}$$

Thus

$$\Delta w_{ji} = -\eta\frac{\partial E}{\partial w_i j} = \eta\delta_j x_{ji} \tag{17}$$

Now consider the case when *j* is a hidden unit. Like before, we make the following two important observations.

1. For each unit *k* downstream from *j*, z_k is a function of z_j

2. The contribution to error by all units $l \neq j$ in the same layer as *j* is independent of w_{ji}

We want to calculate $\frac{\partial E}{\partial w_{ji}}$ for each input weight w_{ji} for each hidden unit *j*. Note that w_{ji} influences just z_j which influences o_j which influences $z_k \forall k \in Downstream(j)$ each of which influence *E*. So we can write

$$\frac{\partial E}{\partial w_{ji}} = \sum_{k \in Downstream(j)} \frac{\partial E}{\partial z_k} \cdot \frac{\partial z_k}{\partial o_j} \cdot \frac{\partial o_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ji}}$$

$$= \sum_{k \in Downstream(j)} \frac{\partial E}{\partial z_k} \cdot \frac{\partial z_k}{\partial o_j} \cdot \frac{\partial o_j}{\partial z_j} \cdot x_{ji}$$

Again note that all the terms except x_{ji} in the above product are the same regardless of which input weight of unit j we are trying to update. Like before, we denote this common quantity by δ_j. Also note that $\frac{\partial E}{\partial z_k} = \delta_k$, $\frac{\partial z_k}{\partial o_j} = w_{kj}$ and $\frac{\partial o_j}{\partial z_j} = o_j(1 - o_j)$. Substituting,

$$\delta_j = \sum_{k \in Downstream(j)} \frac{\partial E}{\partial z_k} \cdot \frac{\partial z_k}{\partial o_j} \cdot \frac{\partial o_j}{\partial z_j}$$

$$= \sum_{k \in Downstream(j)} \delta_k w_{kj} o_j(1 - o_j)$$

Thus,

$$\delta_k = o_j(1 - o_j) \sum_{k \in Downstream(j)} \delta_k w_{kj} \qquad (18)$$

We are now in a position to state the Backpropagation algorithm formally.

Formal statement of the algorithm:

Stochastic Backpropagation (training examples, η, n_i, n_h, n_o)

Each training example is of the form $\langle \vec{x}, \vec{t} \rangle$ where $\vec{x}$ is the input vector and $\vec{t}$ is the target vector. η is the learning rate (e.g., .05). n_i, n_h and n_o are the number of input, hidden and output nodes respectively. Input from unit *i* to unit *j* is denoted x_{ji} and its weight is denoted by w_{ji}.

- Create a feed-forward network with n_i inputs, n_h hidden units, and n_o output units.
- Initialize all the weights to small random values (e.g., between -.05 and .05)
- Until termination condition is met, Do
    - For each training example $\langle \vec{x}, \vec{t} \rangle$, Do
        1. Input the instance $\vec{x}$ and compute the output o_u of every unit.
        2. For each output unit *k*, calculate

$$\delta_k = o_k(1 - o_k)(t_k - o_k)$$

        3. For each hidden unit *h*, calculate

$$\delta_h = o_h(1 - o_h) \sum_{k \in Downstream(h)} w_{kh}\delta_k$$

        4. Update each network weight w_{ji} as follows:

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$
$$\text{where} \quad \Delta w_{ji} = \eta \delta_j x_j i$$

6.9 Summary

In this lesson we have investigated different types of structural knowledge representation methods. We considered associative networks (semantic net) , , a representation based on a structure of linked nodes(concepts) and arcs (relations) connecting the nodes. With these networks we saw how related concepts could be structured into cohesive units and exhibited as graphical representation. A frame is a collection of attributes (usually called slots) and associated values (and possibly constraints on values) that describe some entity in the world. In this lesion we also described a special frame-like structure called scripts. Scripts are used to represent stereotypical patterns for commonly occurring events. Like a play scripts contains actors, roles, props, and scenes, which combine to represent a familiar situation. Scripts have been used in a number of programs, which read and "understood" language in the form of stories.

6.10 Keywords

Probabilistic Reasoning, Use of Certainty Factors, Fuzzy Logic, Concept of Learning, Learning Automata, Genetic Algorithm, Learning by Induction, Neural Networks, Back Propagation Algorithm.

6.11 Check Your Progress

Answer the following questions

Q1. How machine learning distinguished from general knowledge acquisition?

Q2. Describe the role of each component of a general learning model and why it is needed for the learning process.

Q3. Explain why inductive learning should require more inference than learning by being told (instructions).

Q4. Describe the similarities and difference between learning automata and genetic algorithms.

Q5. Write short note on the following: -

d. Probabilistic Reasoning

e. Use of Certainty Factors

f. Fuzzy Logic

g. Neural Network

6.12 Reference/Suggested Reading

- Foundations of Artificial Intelligence and Expert System - V S Janakiraman, K Sarukesi, & P Gopalakrishanan, Macmillan Series.
- Artificial Intelligence – E. Rich and K. Knight
- Principles of Artificial Intelligence – Nilsson
- Expert Systems-Paul Harmon and David King, Wiley Press.
- Rule Based Expert System-Bruce G. Buchanan and Edward H. Shortliffe, eds., Addison Wesley.
- Introduction to Artificial Intelligence and Expert System- Dan W. Patterson, PHI, Feb., 2003.

SUBJECT: ARTIFICIAL INTELLIGENCE	
COURSE CODE: MCA-23	**AUTHOR:**
LESSON NO. 7	**VETTER:**
PLANNING AND NATURAL LANGUAGE PROCESSING	

UNIT STRUCTURE

7.0 Objectives

7.1 Planning

7.2 Planning Components

7.3 Goal Stack Planning

7.4 Nonlinear Planning using Constraint Posting

7.5 Hierarchical Planning

7.6 Understanding

7.7 Natural Language Processing

7.8 Steps in Natural Language Processing

7.9 Game Playing and Applications

7.10 Summary

7.11 Check Your Progress

7.12 Reference/Suggested Reading

7.0 OBJECTIVE

The present lesson elaborates the application of AI i.e. Expert System. Expert System is a program that is expertise in a particular domain. MYCIN and RI are also discussed as case study of an expert system. Upon completion of this lesson students know about distinguish features of an expert system and how to use the existing expert system (i.e. MYCIN & RI).

7.1 PLANNING

BLOCKS WORLD PROBLEM

In order to compare the variety of methods of planning, we should find it useful to look at all of them in a single domain that is complex enough that the need for each of the mechanisms is apparent yet simple enough that easy-to-follow examples can be found.

- There is a flat surface on which blocks can be placed.
- There are a number of square blocks, all the same size.
- They can be stacked one upon the other.
- There is robot arm that can manipulate the blocks.

Actions of the robot arm

1. UNSTACK(A, B): Pick up block A from its current position on block B.
2. STACK(A, B): Place block A on block B.
3. PICKUP(A): Pick up block A from the table and hold it.
4. PUTDOWN(A): Put block A down on the table. Notice that the robot arm can hold only one block at a time. **Predicates**
   - In order to specify both the conditions under which an operation may be performed and the results of performing it, we need the following predicates:

1. ON(A, B): Block A is on Block B.
2. ONTABLES(A): Block A is on the table.
3. CLEAR(A): There is nothing on the top of Block A.
4. HOLDING(A): The arm is holding Block A.
5. ARMEMPTY: The arm is holding nothing.

Robot problem-solving systems (STRIPS)

- List of new predicates that the operator causes to become true is ADD List
- Moreover, List of old predicates that the operator causes to become false is DELETE List
- PRECONDITIONS list contains those predicates that must be true for the operator to be applied.

STRIPS style operators for BLOCKs World

STACK(x, y)

P: CLEAR(y)^HOLDING(x) D: CLEAR(y)^HOLDING(x) A: ARMEMPTY^ON(x, y) UNSTACK(x, y)

PICKUP(x)

P: CLEAR(x) ^ ONTABLE(x) ^ARMEMPTY D: ONTABLE(x) ^ ARMEMPTY

A: HOLDING(x) PUTDOWN(x)

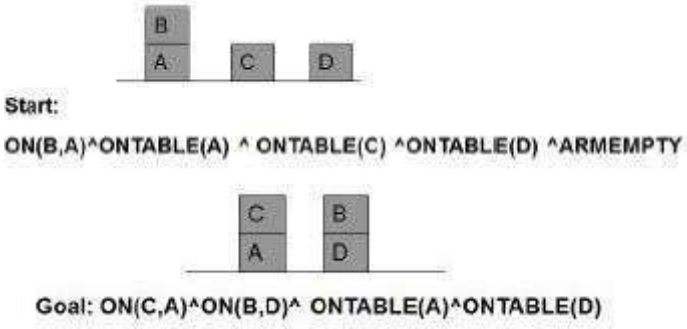
Goal Stack Planning

To start with goal stack is simply:

- ON(C,A)^ON(B,D)^ONTABLE(A)^ONTABLE(D)

This problem is separate into four sub-problems, one for each component of the goal.

Two of the sub-problems ONTABLE(A) and ONTABLE(D) are already true in the initial state.



Alternative 1: Goal Stack:

- ON(C,A)
- ON(B,D)
- ON(C,A)^ON(B,D)^OTAD Alternative 2: Goal stack:
- ON(B,D)
- ON(C,A)
- ON(C,A)^ON(B,D)^OTAD

Exploring Operators

- Pursuing alternative 1, we check for operators that could cause ON(C, A)
- Out of the 4 operators, there is only one STACK. So it yields:
- STACK(C,A)
- ON(B,D)
- ON(C,A)^ON(B,D)^OTAD

- Preconditions for STACK(C, A) should be satisfied, we must establish them as sub-goals:
- CLEAR(A)
- HOLDING(C)
- CLEAR(A)^HOLDING(C)
- STACK(C,A) o ON(B,D)
- ON(C,A)^ON(B,D)^OTAD
- Here we exploit the Heuristic that if HOLDING is one of the several goals to be achieved at once, it should be tackled last.

Goal stack Planning

- Next, we see if CLEAR(A) is true. It is not. The only operator that could make it true is UNSTACK(B, A). Also, This produces the goal stack:
- ON(B, A)
- CLEAR(B)
- ON(B,A)^CLEAR(B)^ARMEMPTY
- UNSTACK(B, A)
- HOLDING(C)
- CLEAR(A)^HOLDING(C)
- STACK(C,A)
- ON(B,D)
- ON(C,A)^ON(B,D)^OTAD
- We see that we can pop predicates on the stack till we reach HOLDING(C) for which we need to find a suitable operator.
- Moreover, The operators that might make HOLDING(C) true: PICKUP(C) and UNSTACK(C, x). Without looking ahead, since we cannot tell which of these operators is appropriate. Also, we create two branches of the search tree corresponding to the following goal stacks:

ALT1:	ALT2:
ONTABLE(C) CLEAR(C) ARMEMPTY ONTABLE(C) ^CLEAR(C)^ARMEMPTY PICKUP(C) CLEAR(A)^HOLDING(C) STACK(C,A) ON(B,D) ON(C,A)^ON(B,D)^OTAD	ON(C, x) CLEAR(C) ARMEMPTY ON(C,x)^CLEAR(C)^ARMEMPTY UNSTACK(C,x) CLEAR(A)^HOLDING(C) STACK(C,A) ON(B,D) ON(C,A)^ON(B,D)^OTAD

Complete plan

1. UNSTACK(C, A)
2. PUTDOWN(C)
3. PICKUP(A)
4. STACK(A, B)
5. UNSTACK(A, B)
6. PUTDOWN(A)
7. PICKUP(B)
8. STACK(B, C)
9. PICKUP(A)
10. STACK(A,B)

7.2 PLANNING COMPONENTS

- Methods which focus on ways of decomposing the original problem into appropriate subparts and on ways of recording and handling interactions among the subparts as they are detected during the problem-solving process are often called as planning.
- Planning refers to the process of computing several steps of a problem-solving procedure before executing any of them.

Components of a planning system

Choose the best rule to apply next, based on the best available heuristic information.

- The most widely used technique for selecting appropriate rules to apply is first to isolate a set of differences between desired goal state and then to identify those rules that are relevant to reduce those differences.

- If there are several rules, a variety of other heuristic information can be exploited to choose among them.

Apply the chosen rule to compute the new problem state that arises from its application.

- In simple systems, applying rules is easy. Each rule simply specifies the problem state that would result from its application.
- In complex systems, we must be able to deal with rules that specify only a small part of the complete problem state.
- One way is to describe, for each action, each of the changes it makes to the state description.

Detect when a solution has found.

- A planning system has succeeded in finding a solution to a problem when it has found a sequence of operators that transform the initial problem state into the goal state.
- How will it know when this has done?
- In simple problem-solving systems, this question is easily answered by a straightforward match of the state descriptions.
- One of the representative systems for planning systems is predicate logic. Suppose that as a part of our goal, we have the predicate P(x).
- To see whether P(x) satisfied in some state, we ask whether we can prove P(x) given the assertions that describe that state and the axioms that define the world model.

Detect dead ends so that they can abandon and the system's effort directed in more fruitful directions.

- As a planning system is searching for a sequence of operators to solve a particular problem, it must be able to detect when it is exploring a path that can never lead to a solution.
- The same reasoning mechanisms that can use to detect a solution can often use for detecting a dead end.
- If the search process is reasoning forward from the initial state. It can prune any path that leads to a state from which the goal state cannot reach.
- If search process reasoning backward from the goal state, it can also terminate a

path either because it is sure that the initial state cannot reach or because little progress made.

Detect when an almost correct solution has found and employ special techniques to make it totally correct.

- The kinds of techniques discussed are often useful in solving nearly decomposable problems.
- One good way of solving such problems is to assume that they are completely decomposable, proceed to solve the sub-problems separately. And then check that when the sub-solutions combined. They do in fact give a solution to the original problem.

7.3 GOAL STACK PLANNING

- Methods which focus on ways of decomposing the original problem into appropriate subparts and on ways of recording. And handling interactions among the subparts as they are detected during the problem-solving process are often called as planning.
- Planning refers to the process of computing several steps of a problem-solving procedure before executing any of them.

Goal Stack Planning Method

- In this method, the problem solver makes use of a single stack that contains both goals and operators. That have proposed to satisfy those goals.
- The problem solver also relies on a database that describes the current situation and a set of operators described as PRECONDITION, ADD and DELETE lists.
- The goal stack planning method attacks problems involving conjoined goals by solving the goals one at a time, in order.
- A plan generated by this method contains a sequence of operators for attaining the first goal, followed by a complete sequence for the second goal etc.
- At each succeeding step of the problem-solving process, the top goal on the stack will pursue.
- When a sequence of operators that satisfies it, found, that sequence applied to the state description, yielding new description.

- Next, the goal that then at the top of the stack explored. And an attempt made to satisfy it, starting from the situation that produced as a result of satisfying the first goal.
- This process continues until the goal stack is empty.
- Then as one last check, the original goal compared to the final state derived from the application of the chosen operators.
- If any components of the goal not satisfied in that state. Then those unsolved parts of the goal reinserted onto the stack and the process resumed.

7.4 NONLINEAR PLANNING USING CONSTRAINT POSTING

- Difficult problems cause goal interactions.
- The operators used to solve one sub-problem may interfere with the solution to a previous sub-problem.
- Most problems require an intertwined plan in which multiple sub-problems worked on simultaneously.
- Such a plan is called nonlinear plan because it is not composed of a linear sequence of complete sub-plans.

Constraint Posting

- The idea of constraint posting is to build up a plan by incrementally hypothesizing operators, partial orderings between operators, and binding of variables within operators.
- At any given time in the problem-solving process, we may have a set of useful operators but perhaps no clear idea of how those operators should order with respect to each other.
- A solution is a partially ordered, partially instantiated set of operators to generate an actual plan. And we convert the partial order into any number of total orders.

Constraint Posting versus State Space search

State Space Search

- Moves in the space: Modify world state via operator

- Model of time: Depth of node in search space
- Plan stored in Series of state transitions Constraint Posting Search
- Moves in the space: Add operators, Oder Operators, Bind variables Or Otherwise constrain plan
- Model of Time: Partially ordered set of operators
- Plan stored in Single node

Algorithm: Nonlinear Planning (TWEAK)

1. Initialize S to be the set of propositions in the goal state.
2. Remove some unachieved proposition P from S.
3. Moreover, Achieve P by using step addition, promotion, DE clobbering, simple establishment or separation.
4. Review all the steps in the plan, including any new steps introduced by step addition, to see if any of their preconditions unachieved. Add to S the new set of unachieved preconditions.
5. Also, If S is empty, complete the plan by converting the partial order of steps into a total order, instantiate any variables as necessary.
6. Otherwise, go to step 2.

7.5 HIERARCHICAL PLANNING

- In order to solve hard problems, a problem solver may have to generate long plans.
- It is important to be able to eliminate some of the details of the problem until a solution that addresses the main issues is found.
- Then an attempt can make to fill in the appropriate details.
- Early attempts to do this involved the use of macro operators, in which larger operators were built from smaller ones.
- In this approach, no details eliminated from actual descriptions of the operators.

ABSTRIPS

A better approach developed in ABSTRIPS systems which actually planned in a hierarchy of abstraction spaces, in each of which preconditions at a lower level of abstraction ignored.

ABSTRIPS approach is as follows:

- First solve the problem completely, considering only preconditions whose criticality value is the highest possible.
- These values reflect the expected difficulty of satisfying the precondition.
- To do this, do exactly what STRIPS did, but simply ignore the preconditions of lower than peak criticality.
- Once this done, use the constructed plan as the outline of a complete plan and consider preconditions at the next-lowest criticality level.
- Augment the plan with operators that satisfy those preconditions.
- Because this approach explores entire plans at one level of detail before it looks at the lower-level details of any one of them, it has called length-first approach.

The assignment of appropriate criticality value is crucial to the success of this hierarchical planning method.

Those preconditions that no operator can satisfy are clearly the most critical.

Example, solving a problem of moving the robot, for applying an operator, PUSH-THROUGH DOOR, the precondition that there exists a door big enough for the robot to get through is of high criticality since there is nothing we can do about it if it is not true.

REACTIVE SYSTEMS

- The idea of reactive systems is to avoid planning altogether, and instead, use the observable situation as a clue to which one can simply react.
- A reactive system must have access to a knowledge base of some sort that describes what actions should be taken under what circumstances.
- A reactive system is very different from the other kinds of planning systems we have discussed. Because it chooses actions one at a time.
- It does not anticipate and select an entire action sequence before it does the first thing.
- The example is a Thermostat. The job of the thermostat is to keep the temperature constant inside a room.
- Reactive systems are capable of surprisingly complex behaviors.
- The main advantage reactive systems have over traditional planners is that they operate robustly in domains that are difficult to model completely and

accurately.

- Reactive systems dispense with modeling altogether and base their actions directly on their perception of the world.
- Another advantage of reactive systems is that they are extremely responsive since they avoid the combinatorial explosion involved in deliberative planning.
- This makes them attractive for real-time tasks such as driving and walking.

Triangle tables

- Provides a way of recording the goals that each operator expected to satisfy as well as the goals that must be true for it to execute correctly.

Meta-planning

- A technique for reasoning not just about the problem solved but also about the planning process itself.

Macro-operators

- Allow a planner to build new operators that represent commonly used sequences of operators.

Case-based planning:

- Re-uses old plans to make new ones.

7.6 UNDERSTANDING

Understanding is the simplest procedure of all human beings. Understanding means ability to determine some new knowledge from a given knowledge. For each action of a problem, the mapping of some new actions is very necessary. Mapping the knowledge means transferring the knowledge from one representation to another representation. For example, if you will say “I need to go to New Delhi” for which you will book the tickets. The system will have “understood” if it finds the first available plane to New Delhi. But if you will say the same thing to you friends, who knows that your family lives in “New Delhi”, he/she will have “understood” if he/she realizes that there may be a problem or occasion in your family. For people, understanding applies to inputs from all the senses. Computer understanding has so far been applied primarily to images, speech and typed languages. It is important to keep in mind that the success or failure of an “understanding” problem can rarely be measured in an absolute sense but must instead be measured with respect to a

particular task to be performed. There are some factors that contribute to the difficulty of an understanding problem.

1. If the target representation is very complex for which you cannot map from the original representation.
2. There are different types of mapping factors may arise like one-to-one, one-to-many and many to many.
3. Some noise or disturbing factors are also there.
4. The level of interaction of the source components may be complex one.
5. The problem solver might be unknown about some more complex problems.
6. The intermediary actions may also be unavailable.

Consider an example of an English sentence which is being used for communication with a keyword based data retrieval system. Suppose I want to know all about the temples in India. So I would need to be translated into a representation such as The above sentence is a simple sentence for which the corresponding representation may be easy to implement. But what for the complex queries?

Consider the following query.

“Ram told Sita he would not eat apple with her. He has to go to the office”.

This type of complex queries can be modeled with the conceptual dependency representation which is more complex than that of simple representation. Constructing these queries is very difficult since more informationare to be extracted. Extracting more information will require some more knowledge. Also the type of mapping process is not quite easy to the problem solver. Understanding is the process of mapping an input from its original form to a more useful one.

The simplest kind of mapping is “one-toone”.

In one-to-one mapping each different problems would lead to only one solution. But there are very few inputs which are one-to-one. Other mappings are quite difficult to implement. Many-to- one mappings are frequent is that free variation is often allowed, either because of the physical limitations of that produces the inputs or because such variation simply makes the task of generating the inputs.

Many to one mapping require that the understanding system know about all the ways that a target representation can be expressed in the source language. One-to-many

mapping requires a great deal of domain knowledge in order to make the correct choice among the available target representation.

The mapping process is simplest if each component can be mapped without concern for the other components of the statement. If the number of interactions increases, then the complexity of the problem will increase. In many understanding situations the input to which meaning should be assigned is not always the input that is presented to the under stander.

Because of the complex environment in which understanding usually occurs, other things often interfere with the basic input before it reaches the under stander. Hence the understanding will be more complex if there will be some sort of noise on the inputs.

7.7 NATURAL LANGUAGE PROCESSING

Language meant for communicating with the world. Also, by studying language, we can come to understand more about the world. If we can succeed at building computational mode of language, we will have a powerful tool for communicating with the world. Also, we look at how we can exploit knowledge about the world, in combination with linguistic facts, to build computational natural language systems.

Natural Language Processing (NLP) problem can divide into two tasks:

1. Processing written text, using lexical, syntactic and semantic knowledge of the language as well as the required real-world information.
2. Processing spoken language, using all the information needed above plus additional knowledge about phonology as well as enough added information to handle the further ambiguities that arise in speech.

7.8 STEPS IN NATURAL LANGUAGE PROCESSING

Morphological Analysis

- Individual words analyzed into their components and non-word tokens such as punctuation separated from the words.

Syntactic Analysis

- Linear sequences of words transformed into structures that show how the words relate to each other.
- Moreover, Some word sequences may reject if they violate the language's rule

for how words may combine.

Semantic Analysis

- The structures created by the syntactic analyzer assigned meanings.
- Also, A mapping made between the syntactic structures and objects in the task domain.
- Moreover, Structures for which no such mapping possible may reject. Discourse integration
- The meaning of an individual sentence may depend on the sentences that precede it. And also, may influence the meanings of the sentences that follow it.

Pragmatic Analysis

- Moreover, The structure representing what said reinterpreted to determine what was actually meant.

Summary

- Results of each of the main processes combine to form a natural language system.
- All of the processes are important in a complete natural language understanding system.
- Not all programs are written with exactly these components.
- Sometimes two or more of them collapsed.
- Doing that usually results in a system that is easier to build for restricted subsets of English but one that is harder to extend to wider coverage.

STEPS IN NATURAL LANGUAGE PROCESSING

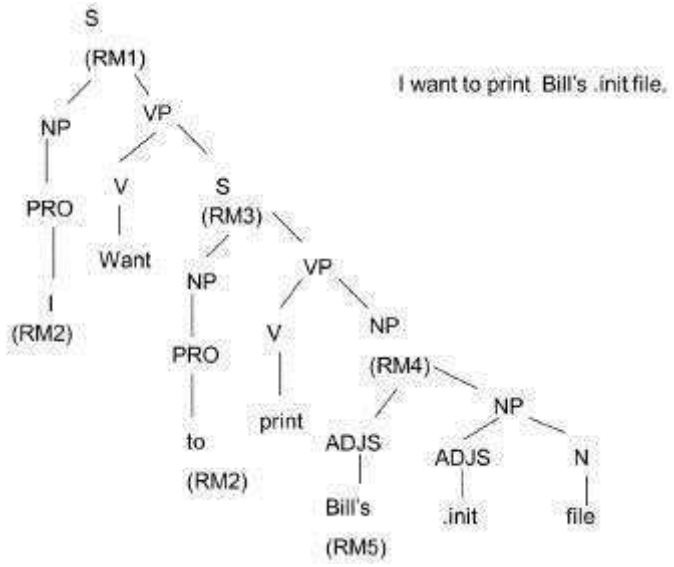
Morphological Analysis

- Suppose we have an English interface to an operating system and the following sentence typed: I want to print Bill's .init file.
- The morphological analysis must do the following things:
- Pull apart the word "Bill's" into proper noun "Bill" and the possessive suffix "'s"
- Recognize the sequence ".init" as a file extension that is functioning as an adjective in the sentence.

- This process will usually assign syntactic categories to all the words in the sentence.

Syntactic Analysis

- A syntactic analysis must exploit the results of the morphological analysis to build a structural description of the sentence.
- The goal of this process, called parsing, is to convert the flat list of words that form the sentence into a structure that defines the units that represented by that flat list.
- The important thing here is that a flat sentence has been converted into a hierarchical structure. And that the structure corresponds to meaning units when a semantic analysis performed.
- Reference markers (set of entities) shown in the parenthesis in the parse tree.
- Each one corresponds to some entity that has mentioned in the sentence.
- These reference markers are useful later since they provide a place in which to accumulate information about the entities as we get it.



Semantic Analysis

The semantic analysis must do two important things:

1. It must map individual words into appropriate objects in the knowledge base or database.
2. It must create the correct structures to correspond to the way the meanings of the individual words combine with each other.

Discourse Integration

- Specifically, we do not know whom the pronoun “I” or the proper noun “Bill” refers to.
- To pin down these references requires an appeal to a model of the current discourse context, from which we can learn that the current user is USER068 and that the only person named “Bill” about whom we could be talking is USER073.
- Once the correct referent for Bill known, we can also determine exactly which file referred to.

Pragmatic Analysis

- The final step toward effective understanding is to decide what to do as a result.
- One possible thing to do to record what was said as a fact and done with it.
- For some sentences, a whose intended effect is clearly declarative, that is the precisely correct thing to do.
- But for other sentences, including this one, the intended effect is different.
- We can discover this intended effect by applying a set of rules that characterize cooperative dialogues.
- The final step in pragmatic processing to translate, from the knowledge-based representation to a command to be executed by the system.

SYNTACTIC PROCESSING

- Syntactic Processing is the step in which a flat input sentence converted into a hierarchical structure that corresponds to the units of meaning in the sentence. This process called parsing.
- It plays an important role in natural language understanding systems for two reasons:

1. Semantic processing must operate on sentence constituents. If there is no syntactic parsing step, then the semantics system must decide on its own constituents. If parsing is done, on the other hand, it constrains the number of constituents that semantics can consider.
2. Syntactic parsing is computationally less expensive than is semantic processing. Thus it can play a significant role in reducing overall system complexity.

- Although it is often possible to extract the meaning of a sentence without using grammatical facts, it is not always possible to do so.
- Almost all the systems that are actually used have two main components:

1. A declarative representation, called a grammar, of the syntactic facts about the language.
2. A procedure, called parser that compares the grammar against input sentences to produce parsed structures.

Grammars and Parsers

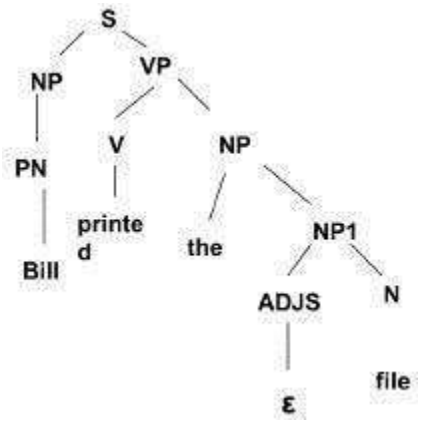
- The most common way to represent grammars is a set of production rules.
- The first rule can read as "A sentence composed of a noun phrase followed by Verb Phrase"; the Vertical bar is OR; ε represents the empty string.
- Symbols that further expanded by rules called non-terminal symbols.
- Symbols that correspond directly to strings that must found in an input sentence called terminal symbols.
- Grammar formalism such as this one underlies many linguistic theories, which in turn provide the basis for many natural language understanding systems.
- Pure context-free grammars are not effective for describing natural languages.
- NLPs have less in common with computer language processing systems such as compilers.
- Parsing process takes the rules of the grammar and compares them against the input sentence.
- The simplest structure to build is a Parse Tree, which simply records the rules and how they matched.
- Every node of the parse tree corresponds either to an input word or to a non-terminal in our grammar.
- Each level in the parse tree corresponds to the application of one grammar rule.

Example for Syntactic Processing – Augmented Transition Network

Syntactic Processing is the step in which a flat input sentence is converted into a hierarchical structure that corresponds to the units of meaning in the sentence. This process called parsing. It plays an important role in natural language understanding systems for two reasons:

1. Semantic processing must operate on sentence constituents. If there is no syntactic parsing step, then the semantics system must decide on its own constituents. If parsing is done, on the other hand, it constrains the number of constituents that semantics can consider.
2. Syntactic parsing is computationally less expensive than is semantic processing. Thus it can play a significant role in reducing overall system complexity.

Example: A Parse tree for a sentence: Bill Printed the file



The grammar specifies two things about a language:

1. Its weak generative capacity, by which we mean the set of sentences that contained within the language. This set made up of precisely those sentences that can completely match by a series of rules in the grammar.
2. Its strong generative capacity, by which we mean the structure to assign to each grammatical sentence of the language.

Augmented Transition Network (ATN)

- An augmented transition network is a top-down parsing procedure that allows various kinds of knowledge to incorporated into the parsing system so it can operate efficiently.
- ATNs build on the idea of using finite state machines (Markov model) to parse sentences.
- Instead of building an automaton for a particular sentence, a collection of transition graphs built.
- A grammatically correct sentence parsed by reaching a final state in any state graph.

- Transitions between these graphs simply subroutine calls from one state to any initial state on any graph in the network.
- A sentence determined to be grammatically correct if a final state reached by the last word in the sentence.
- The ATN is similar to a finite state machine in which the class of labels that can attach to the arcs that define the transition between states has augmented.

Arcs may label with:

- Specific words such as “in’.
- Word categories such as noun.
- Procedures that build structures that will form part of the final parse.
- Procedures that perform arbitrary tests on current input and sentence components that have identified.

Semantic Analysis

- The structures created by the syntactic analyzer assigned meanings.
- A mapping made between the syntactic structures and objects in the task domain.
- Structures for which no such mapping is possible may rejected.
- The semantic analysis must do two important things:
- It must map individual words into appropriate objects in the knowledge base or database.
- It must create the correct structures to correspond to the way the meanings of the individual words combine with each other. Semantic Analysis AI
- Producing a syntactic parse of a sentence is only the first step toward understanding it.
- We must produce a representation of the meaning of the sentence.
- Because understanding is a mapping process, we must first define the language into which we are trying to map.
- There is no single definitive language in which all sentence meaning can describe.
- The choice of a target language for any particular natural language understanding program must depend on what is to do with the meanings once they constructed.

- Choice of the target language in Semantic Analysis AI
- There are two broad families of target languages that used in NL systems, depending on the role that the natural language system playing in a larger system:
- When natural language considered as a phenomenon on its own, as for example when one builds a program whose goal is to read the text and then answer questions about it. A target language can design specifically to support language processing.
- When natural language used as an interface language to another program (such as a db query system or an expert system), then the target language must legal input to that other program. Thus the design of the target language driven by the backend program.

Discourse and Pragmatic Processing

To understand a single sentence, it is necessary to consider the discourse and pragmatic context in which the sentence was uttered.

There are a number of important relationships that may hold between phrases and parts of their discourse contexts, including:

1. Identical entities. Consider the text:
   - Bill had a red balloon. o John wanted it.
   - The word “it” should identify as referring to the red balloon. These types of references called anaphora.
2. Parts of entities. Consider the text:
   - Sue opened the book she just bought.
   - The title page was torn.
   - The phrase “title page” should be recognized as part of the book that was just bought.
3. Parts of actions. Consider the text:
   - John went on a business trip to New York.
   - He left on an early morning flight.
   - Taking a flight should recognize as part of going on a trip.
4. Entities involved in actions. Consider the text:

- My house was broken into last week.
- Moreover, They took the TV and the stereo.
- The pronoun “they” should recognize as referring to the burglars who broke into the house.

5. Elements of sets. Consider the text:
   - The decals we have in stock are stars, the moon, item and a flag.
   - I’ll take two moons.
   - Moons mean moon decals.
6. Names of individuals:
   - Dev went to the movies.
7. Causal chains
   - There was a big snow storm yesterday.
   - So, The schools closed today.
8. Planning sequences:
   - Sally wanted a new car
   - She decided to get a job.
9. Implicit presuppositions:
   - Did Joe fail CS101?

The major focus is on using following kinds of knowledge:

- The current focus of the dialogue.
- Also, A model of each participant’s current beliefs.
- Moreover, The goal-driven character of dialogue.
- The rules of conversation shared by all participants.

Statistical Natural Language Processing

Formerly, many language-processing tasks typically involved the direct hand coding of rules, which is not in general robust to natural-language variation. The machine-learning

paradigm calls instead for using statistical inference to automatically learn such rules through the analysis of large corpora of typical real-world examples (a corpus (plural, "corpora") is a set of documents, possibly with human or computer annotations).

Many different classes of machine learning algorithms have been applied to natural-language processing tasks. These algorithms take as input a large set of "features" that are generated from the input data. Some of the earliest-used algorithms, such as decision trees, produced systems of hard if-then rules similar to the systems of hand-written rules that were then common.

Increasingly, however, research has focused on statistical models, which make soft, probabilistic decisions based on attaching real-valued weights to each input feature. Such models have the advantage that they can express the relative certainty of many different possible answers rather than only one, producing more reliable results when such a model is included as a component of a larger system.

Systems based on machine-learning algorithms have many advantages over hand-produced rules

7.9 Game Playing and Applications

Ever since the advent of Artificial Intelligence (AI), game playing has been one of the most interesting applications of AI. The first chess programs were written by Claude Shannon... Ever since the advent of Artificial Intelligence (AI), game playing has been one of the most interesting applications of AI.

The first chess programs were written by Claude Shannon and by Alan Turing in 1950, almost as soon as the computers became programmable. Games such as chess, tic-tac-toe, and Go are interesting because they offer a pure abstraction of the competition between the two armies.

It is this abstraction which makes game playing an attractive area for AI research.

What is the Minimax algorithm?

Minimax is a recursive algorithm which is used to choose an optimal move for a player assuming that the other player is also playing optimally.

It is used in games such as tic-tac-toe, go, chess, Isola, checkers, and many other two-player games. Such games are called games of perfect information because it is possible to see all the possible moves of a particular game.

There can be two-player games which are not of perfect information such as Scrabble because the opponent's move cannot be predicted. It is similar to how we think when we play a game: "if I make this move, then my opponent can only make only these moves," and so on.

Minimax is called so because it helps in minimizing the loss when the other player chooses the strategy having the maximum loss.

Terminology

- **Game Tree**: It is a structure in the form of a tree consisting of all the possible moves which allow you to move from a state of the game to the next state.

A game can be defined as a search problem with the following components:

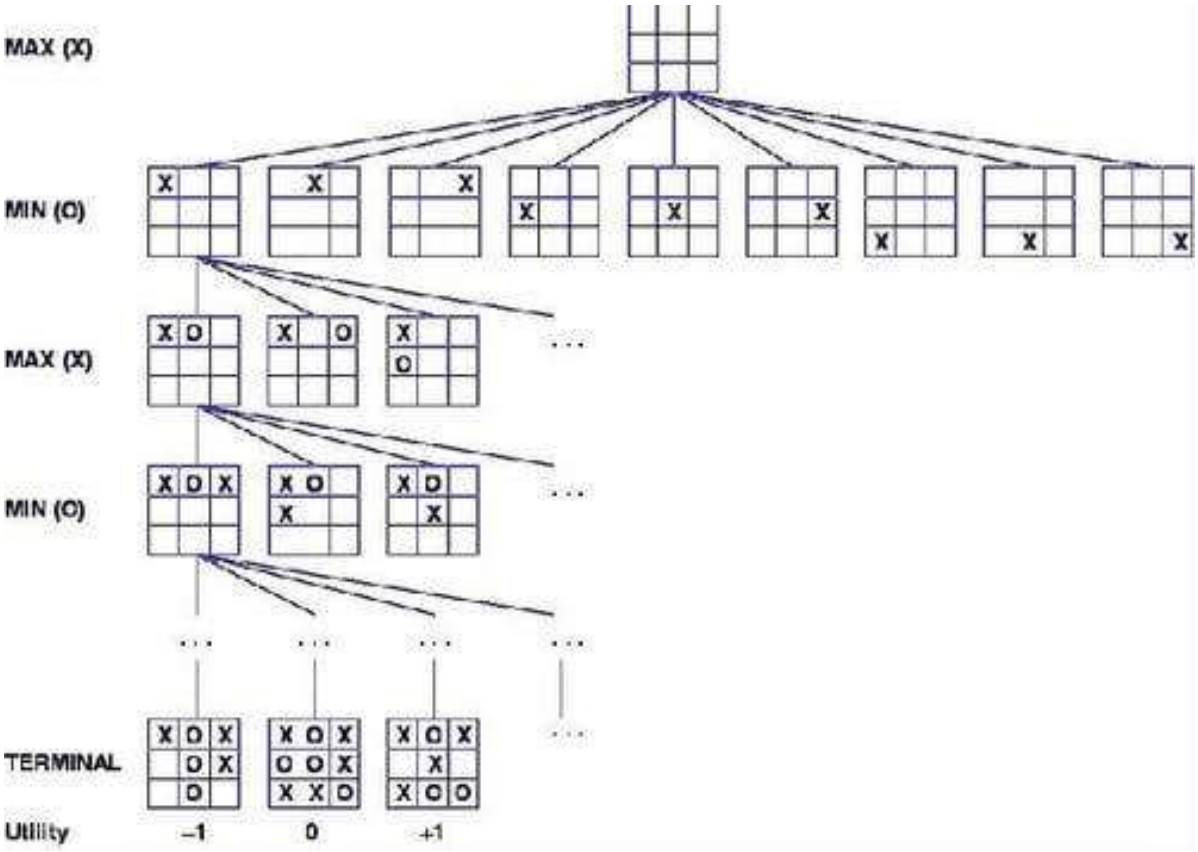
- **Initial state**: It comprises the position of the board and showing whose move it is.
- **Successor function**: It defines what the legal moves a player can make are.
- **Terminal state**: It is the position of the board when the game gets over.
- **Utility function**: It is a function which assigns a numeric value for the outcome of a game. For instance, in chess or tic-tac-toe, the outcome is either a win, a loss, or a draw, and these can be represented by the values +1, -1, or 0, respectively. There are games that have a much larger range of possible outcomes; for instance, the utilities in backgammon varies from +192 to -192. A utility function can also be called a payoff function.

How does the algorithm work?

There are two players involved in a game, called MIN and MAX. The player MAX tries to get the highest possible score and MIN tries to get the lowest possible score, i.e., MIN and MAX try to act opposite of each other.

The general process of the Minimax algorithm is as follows:

Step 1: First, generate the entire game tree starting with the current position of the game all the way up to the terminal states. This is how the game tree looks like for the game tic-tac-toe.



Step 2: Apply the utility function to get the utility values for all the terminal states.

Step 3: Determine the utilities of the higher nodes with the help of the utilities of the terminal nodes.

Step 4: Calculate the utility values with the help of leaves considering one layer at a time until the root of the tree.

Step 5: Eventually, all the backed-up values reach to the root of the tree, i.e., the topmost point. At that point, MAX has to choose the highest value.

Therefore, the best opening move for MAX is the left node (or the red one). This move is called the minimax decision as it maximizes the utility following the assumption that the opponent is also playing optimally to minimize it.

7.10 Summary

An expert system is a set of programs that manipulate encoded knowledge to solve problems in a specialized domain that normally requires human expertise. An expert system is usually built with the aid of one or more experts, who must be willing to spend a great deal of effort transferring their expertise to the system. Expert systems

are complex AI programs. However, the expert systems knowledge must be obtained from specialists or other sources of expertise, such as texts, journals articles, and databases.

Knowledge acquisition is the most important aspect of the expert system development. There are three basics approaches of knowledge acquisition i.e. interviewing expert, learning by being told & learning by observation. Knowledge acquisition has five stages throughout the development starting from identification, conceptualization, formalization through implementation & testing

MYCIN is an expert system, which diagnoses infectious blood diseases and determines a recommended list of therapies for the patient. RI (sometimes also called XCON) is a program that configures DEC VAX systems

7.11 Key words

Expert System, Learning, Knowledge Acquisition, MYCIN & RI.

7.12 Check Your Progress

Answer the following questions.

1. What is expert system? Explain the various stages of Expert System.
2. What is knowledge Acquisition? What is its role in AI?
3. Differentiate between RI & MYCIN.

7.13 Reference/Suggested Readings

- ✓ Artificial Intelligence – E. Rich and K. Knight
- ✓ Principles of Artificial Intelligence – Nilsson
- ✓ Expert Systems-Paul Harmon and David King, Wiley Press.
- ✓ Rule Based Expert System-Bruce G. Buchanan and Edward H. Shortliffe, eds., Addison Wesley.