Paper Code : MCA 402 **Author :Om Parkash**
Lesson No. 01 **Vettor : Saroj**
Lesson Name: Scope of Artificial Intelligence

Structure

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 means 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 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.

1.7 Key Words

Artificial Intelligence (AI), Games, Theorem Proving, Vision and Processing, Natural Language Processing, Robotics, Expert System, Search Knowledge.

1.8 Self Assessment Questions (SAQ)

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

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.

Paper Code : MCA 402 **Author :Om Parkash**
Lesson No. 02 **Vettor : Saroj**
Lesson Name: Problem Solving

Structure

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.

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.

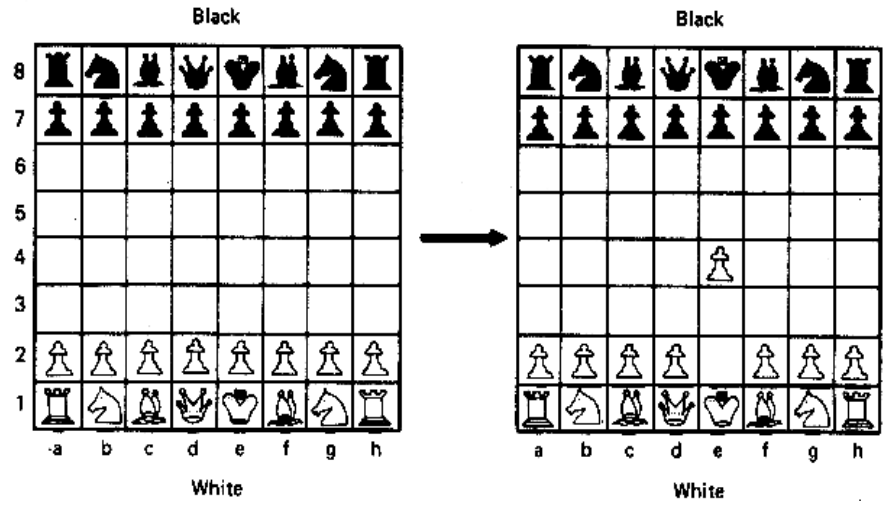


Figure 2.1: One Legal Chess Move

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.

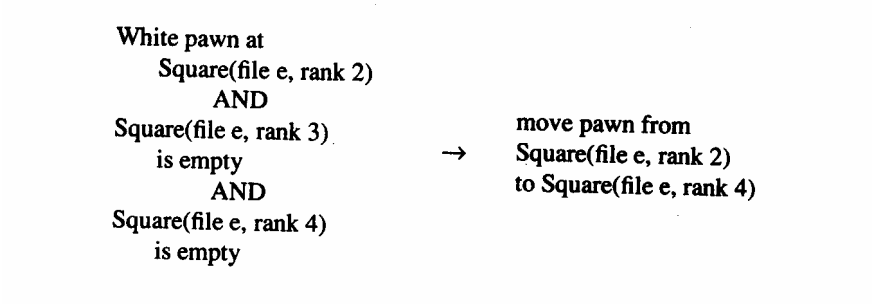


Figure 2.2: Another Way to Describe Chess Moves

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 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 and ACT*
- 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], 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.

2.4 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.

2.5 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.

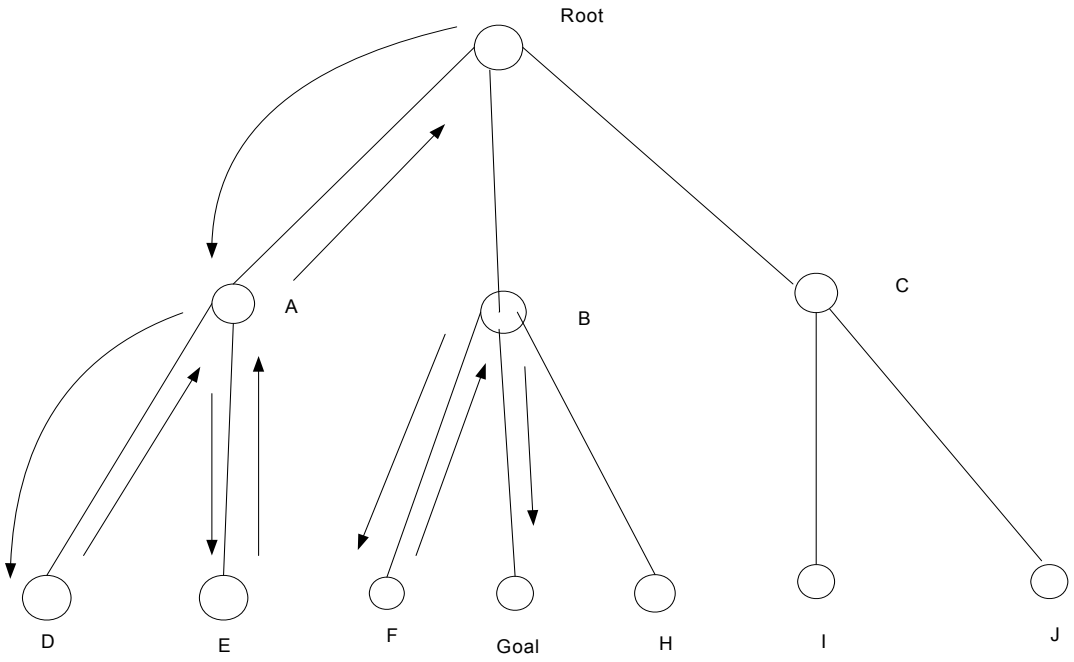


Fig: 2.3 Search tree for Depth-first search

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.

2.5 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.

Fig. 2.4 gives the search tree generated by a breadth-first search.

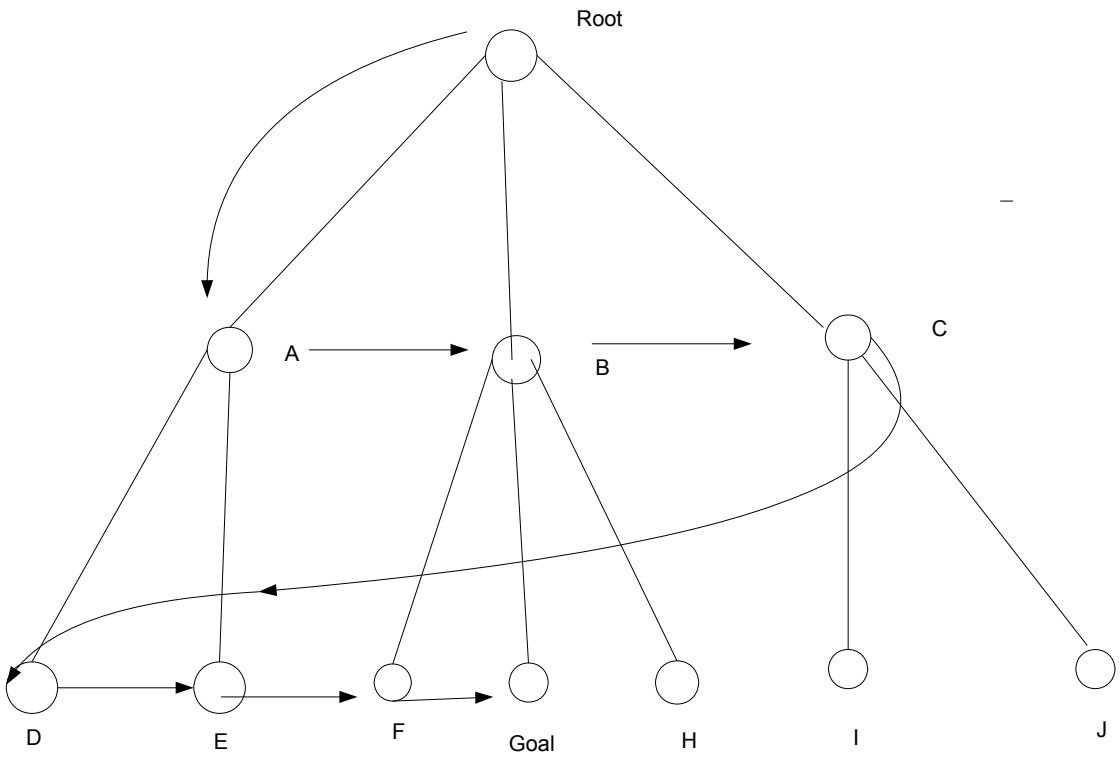


Fig: 2.4 Search tree for Breadth-first search

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.

2.6 Heuristic 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

2.7 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.

The algorithm for hill-climbing Fig. 2.5

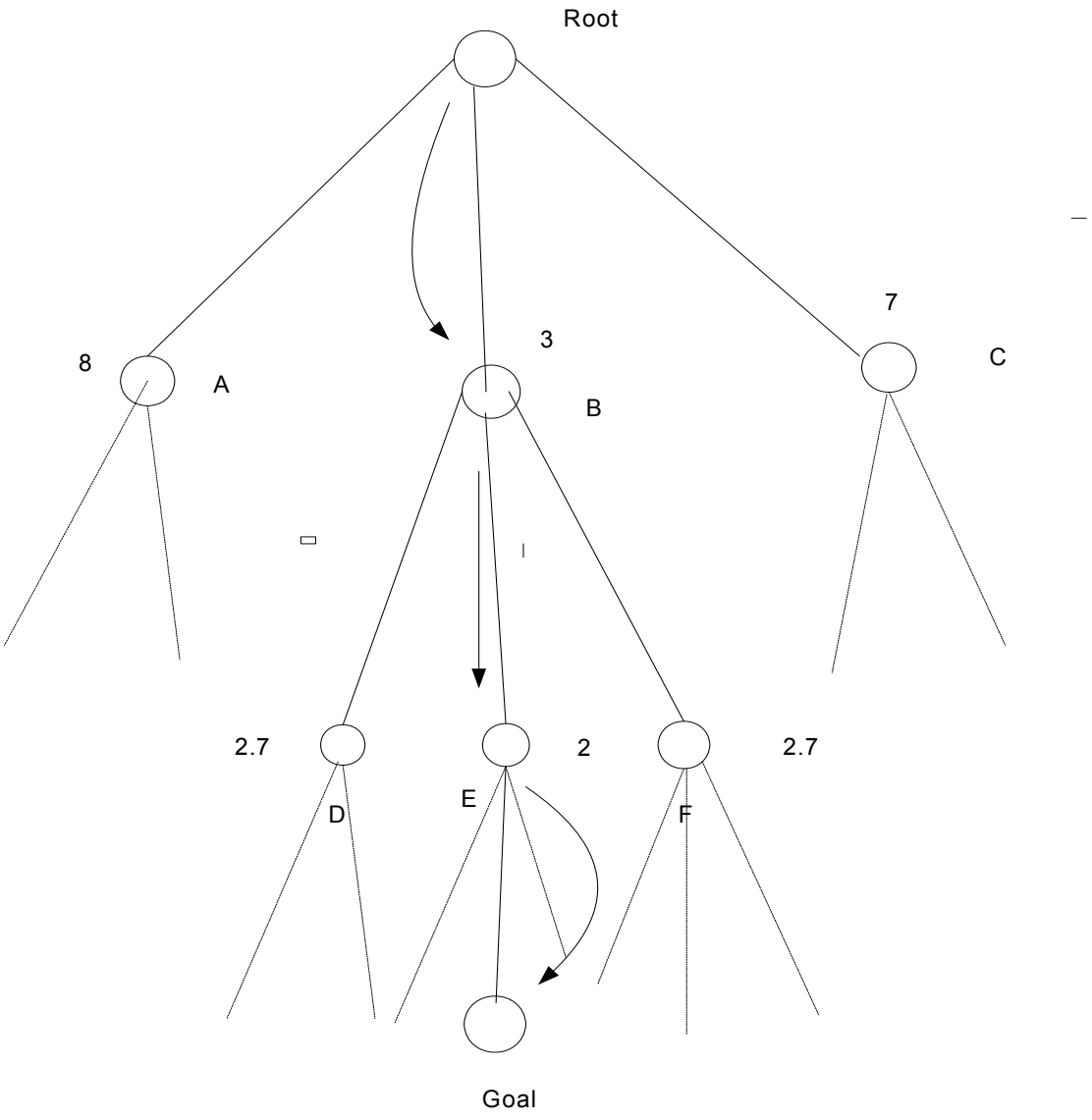


Fig. 2.5 Search tree for hill-climbing procedure

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.

2.8 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.

2.9 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, (e) sort the paths on the queue with lowest-cost paths in front.
2. Otherwise, exit with failure.

2.10 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.

2.11 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 I 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.

2.12 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.

2.13 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.

2.14 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.

2.15 Self-assessment questions

Answer the following questions: -

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

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.

Structure

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 The Role of Logic

The use of symbolic logic to represent knowledge is not new in that it predates the modern computer by a number of decades. Logic is a formal method of reasoning. Many concepts, which can be verbalized, can be translated into symbolic representations, which closely approximate the meaning of these concepts. These symbolic structures can then be manipulated in programs to deduce various facts, to carry out a form of automated reasoning. Logic can be defined as a scientific study of the process of reasoning and the system of rules and procedures that help in the reasoning process. Basically the logic process takes in some information (called premises) and procedures some outputs (called conclusions). Today, First Order Logic (FOPL) or Predicate Logic as it is sometimes called, has assumed one of the most important roles in AI for the representation of knowledge. It is commonly used in program designs and widely discussed in the literature. To understand many of the AI articles and research papers requires comprehensive knowledge of FOPL as well as some related logics.

3.2 Predicate Logic or First Order Logic

A familiarity with Predicate Logic is important to the student of AI for several reasons.

- ✓ Logic offers the only formal approach to reasoning that has a sound theoretical foundation. It is important in our attempts to mechanize or automate the reasoning process in that inference should be correct and logically sound.
- ✓ The structure of FOPL is flexible enough to permit the accurate representation of natural language reasonably well. This is too important in AI system since most knowledge must originate with and be consumed by humans.
- ✓ FOPL is widely accepted by the workers in the AI field as one of the most useful representation methods.

The propositional logic is not powerful enough to represent all types of assertions that are used in computer science and mathematics, or to express certain types of relationship between propositions such as equivalence.

For example, **the assertion "x is greater than 1", where x is a variable, is not a proposition because you can not tell whether it is true or false unless you know the value of x. Thus the prepositional logic cannot deal with such sentences. However, such assertions appear quite often in mathematics and we want to do inferencing on those assertions.**

Also **the pattern involved in the following logical equivalences cannot be captured by the propositional logic:**

"Not all birds fly" is equivalent to "Some birds don't fly".
"Not all integers are even" is equivalent to "Some integers are not even".
"Not all cars are expensive" is equivalent to "Some cars are not expensive",

Each of those propositions is treated independently of the others in propositional logic. For example, if P represents "Not all birds fly" and Q represents "Some integers are not even", then there is no mechanism in propositional logic to find out the P is equivalent to Q. Hence to be used in inferencing, each of these equivalences must be listed individually rather than dealing with a general formula that covers all these equivalences collectively and instantiating it as they become necessary, if only propositional logic is used.
Thus we need more powerful logic to deal with these and other problems. The predicate logic is one of such logic and it addresses these issues among others.

3.3 Unification Algorithm

Unification algorithm is the process of identifying unifiers. Unifier is a substitution that makes two clauses resolvable. The unification algorithm tries to find out the Most General unifier (MGU) between a given set of atomic formulae.

In prepositional 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 man(John) and ¬man(Spot) is not. Thus, in order to determine contradictions, we need a matching procedure that compares two literals are discovers whether there exists a set of substitutions that makes, them identical. There is a straightforward recursive procedure, called the unification algorithm that does just this.

The basic idea of unification is very simple. To attempt to unify two literals, we first check if their initial predicate symbols are the same. If so, we can proceed. Otherwise, there is no way they can be unified, regardless of their arguments. For example, the two literals

– tryassassinate(Marcus, Caesar)

– hate(Marcus, Caesar)

cannot be unified. If the predicate symbols match, then we must check the arguments, one pair at a time. If the first matches, we can continue with the second, and so on. To test each argument pair, we can simply call the unification procedure recursively. The matching rules are simple. Different constants or predicates cannot match; identical ones can. A variable can match another variable, any constant, or a predicate expression, with the restriction that the predicate expression must not contain any instances of the variable being matched.

The only complication in this procedure is that we must find a single, consistent substitution for the entire literal, not separate ones for each piece of it. To do this, we must take each substitution that we find and apply it to the remainder of the literals before we continue trying to unify them. For example, suppose we want to unify the expressions

P(x, x)

P(y, z)

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

y/x

But now, if we simply continue and match x and z, we produce the substitution z/x.

But we cannot substitute both y and z for x, so we have not produced a consistent substitution. What we need to do after finding the first substitution y/x is to make that substitution throughout the literals, giving

P(y,y)

P(y, z)

Now we can attempt to unify arguments y and z, which succeeds with the substitution z/y. The entire unification process has now succeeded with a substitution that is the composition of the two substitutions we found. We write the composition as

(z/y)(y/x)

following standard notation for function composition. In general, the substitution (a1/a2, a3/a4,….)(b1/b2, b3/b4,...)... means to apply all the substitutions of the right-most list, then take the result and apply all the ones of the next list, and so forth, until all substitutions have been applied.

The object of the unification procedure is to discover at least one substitution that causes two literals to match. Usually, if there is one such substitution there are many. For example, the literals

hate(x,y)

hate(Marcus,z)

could be unified with any of the following substitutions:

(Marcus/x,z/y)

(Marcus/x,y/z)

(Marcus/x,Caesar/y,Caesar/z)

(Marcus/x,Polonius/y,Polonius/z)

The first two of these are equivalent except for lexical variation. But the second two, although they produce a match, also produce a substitution that is more restrictive than absolutely necessary for the match. Because the final substitution produced by the, unification process will be used by the resolution procedure, it is useful to generate the most general unifier possible. The algorithm shown below will do that.

Having explained the operation of the unification algorithm, we can now state it concisely. We describe a procedure Unify(L1, L2), which returns as its value a list representing the composition of the substitutions that were performed during the match. The empty list, NIL, indicates that a match was found without any substitutions. The list consisting of the single value FAIL indicates that the unification procedure failed.

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. 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 andL2 are not identical, then return {FAIL}.

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

4. Set SUBST to NIL. (At the end of this procedure, SUBST will contain all the substitutions used to unify L1 and L2.)

5. For i ← 1 to number of arguments in L1:

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

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

   c. If S is not equal to NIL then:

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

      ii. SUBST:= APPEND(S, SUBST)

6. Return SUBST.

The only part of this algorithm that we have not yet discussed is the check in steps 1(b) and l(c) to make sure that an expression involving a given variable is not unified Y with that variable. Suppose we were attempting to unify the expressions

f(x, x)

f(g(x), g(x))

If we accepted g(x) as a substitution for x, then we would have to substitute it for x in the remainder of the expressions. But this leads to infinite recursion since it will never be possible to eliminate x.
Unification has deep mathematical roots and is a useful operation in many AI programs, or example, theorem proverbs and natural language parsers. As a result, efficient data structures and algorithms for unification have been developed.

3.4 Modus Pones

Modus Pones is a property of propositions that is useful in resolution. It can be represented as follows:

$P \, and \, P \rightarrow Q \Rightarrow Q$

Where P and Q are two clauses.

For example
Given: (Joe is a father)
And: (Joe is father) $\rightarrow$ (Joe has child)
Conclude: (Joe has a child)

3.5 Resolution

Robinson in 1965 introduced the resolution principle, which can be directly applied to any set of clauses. The principal is

"Given any two clauses A and B, if there is a literal P1 in A which has a complementary literal P2 in B, delete P1 & P2 from A and B and construct a disjunction of the remaining clauses. The clause so constructed is called resolvent of A and B."

For example, consider the following clauses

A: P V Q V R

B: `p V Q V R

C: `Q V R

Clause A has the literal P which is complementary to `P in B. Hence both of them deleted and a resolvent (disjunction of A and B after the complementary clauses are removed) is generated. That resolvent has again a literal Q whose negation is available in C. Hence resolving those two, one has the final resolvent.

A: P V Q V R (given in the problem)

B: `p V Q V (given in the problem)

D: Q V R (resolvent of A and B)

C: `Q V R (given in the problem)

E: R (resolvent of C and D)

3.6 Dependency Directed Backtracking

If we take a depth-first approach to nonmonotonic reasoning, then the following scenario is likely to occur often: We need to know a fact, F, which cannot be derived monotonically from what we already know, but which can be derived by making some assumption A which seems plausible. So we make assumption A, derive F, and then derive some additional facts G and H from F. We later derive some other facts M and N, but they are completely independent of A and F. A little while later, a new fact comes in that invalidates A. We need to rescind our proof of F, and also our proofs of G and H since they depended on F. But what

about M and N? They didn't depend on F, so there is no logical need to invalidate them. But if we use a conventional backtracking scheme, we have to back up past conclusions in the other in which we derived them. So we have to backup past M and N, thus undoing them, in order to get back to F, G, H and A. To get around this problem, we need a slightly different notion of backtracking, one that is based on logical dependencies rather than the chronological order in which decisions were made. We call this new method dependency-directed backtracking in contrast to chronological backtracking, which we have been using up until now.

Before we go into detail on how dependency-directed backtracking works, it is worth pointing out that although one of the big motivations for it is in handling nonmonotonic reasoning, it turns out to be useful for conventional search programs as well. This is not too surprising when you consider, what any depth-first search program does is to "make a guess" at something, thus creating a branch in the search space. If that branch eventually dies out, then we know that at least one guess that led to it must be wrong. It could be any guess along the branch. In chronological backtracking we have to assume it was the most recent guess ad back up there to try an alternative. Sometimes, though, we have additional information that tells us which guess caused the problem. We'd like to retract only that guess and the work that explicitly depended on it, leaving everything else that has happened in the meantime intact. This is exactly what dependency-directed backtracking does.

As an example, suppose we want to build a program that generates a solution to a fairly simple problem, such as finding a time at which three busy people can all attend a meeting. One way to solve such a problem is first to make an assumption that the meeting will be held on some particular day, say Wednesday, add to the database an assertion to that effect, suitably tagged as an assumption, and then proceed to find a time, checking along the way for any inconsistencies in people's schedules. If a conflict arises, the statement representing the assumption must be discarded and replaced by another, hopefully noncontradictory, one. But, of course, any statements that have been generated along the way that depend on the now-discarded assumption must also be discarded.

Of course, this kind of situation can be handled by a straightforward tree search with chronological backtracking. All assumptions, as well as the inferences drawn from them, are recorded at the search node that created them. When a node is determined to represent a contradiction, simply backtrack to the next node from which there remain unexplored paths. The assumptions and their inferences will disappear automatically. The drawback to this approach is illustrated in Figure 3.1, which shows part of the search tree of a program that is trying to schedule a meeting. To do so, the program must solve a constraints satisfaction problem to find a day and time at which none of the participants is busy and at which there is a sufficiently large room available.

In order to solve the problem, the system must try to satisfy one constraint at a time. Initially, there is little reason to choose one alternative over another, so it decides to schedule the meeting on Wednesday. That creates a new constraint that must be met by the rest of the solution. The assumption that the meeting will be held on Wednesday is stored at the node it generated. Next the program tries to select a time at which all participants are available. Among them, they have regularly scheduled daily meetings at all times except 2:00. So 2:00 is chosen as the meeting time. But it would not have mattered which day was chosen. Then the program discovers that on Wednesday there have no rooms available. So it backtracks past the assumption that the day would be Wednesday and tries another day, Tuesday. Now it must duplicate the chain of reasoning that led it to choose 2:00 as the time because that reasoning was lost when it backtracked to reduce the choice of day. This occurred even though that reasoning did not depend in any way on the assumption that the day would be Wednesday. By withdrawing statements based on the order which they were generated by the search process rather than on the basis of responsibility for inconsistency, we may waste a great deal of effort.

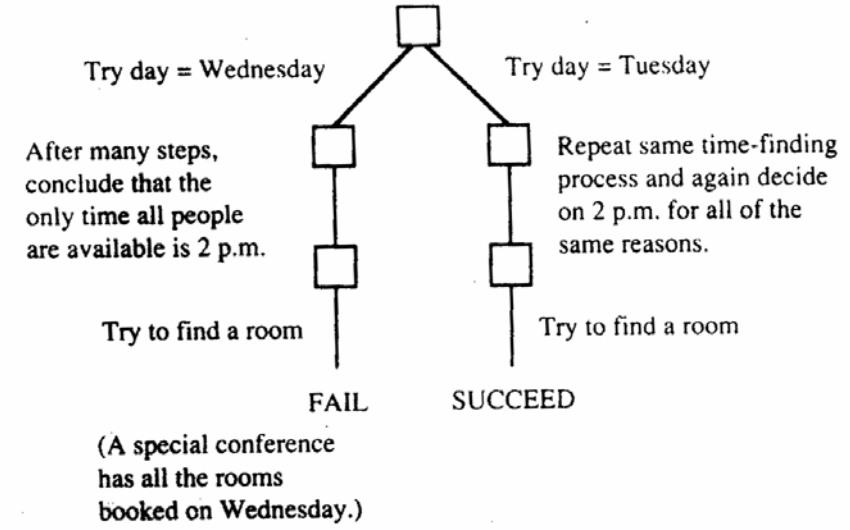


Figure 3.1: Nondependency-Directed Backtracking

If we want to use dependency-directed backtracking instead, so that we do not waste this effort, then we need to do the following things:

Associate with each node one or more justifications. Each justification corresponds to a derivation process that led to the node. (Since it is possible to derive the same node in several different ways, we want to allow for the possibility of multiple justifications). Each justification must contain a list of all the nodes (facts, rules, assumptions) on which its derivation depended.

Provide a mechanism that, when given a contradiction node and its justification, computes the “no-good” set of assumptions that underline the justification. The no-good set is defined to be the minimal set of assumptions such that if you

remove any element from the set, the justification will no longer be valid and the inconsistent node will no longer be believed.

Provide a mechanism for considering a no-good set and choosing an assumption to retract.

Provide a mechanism for propagating the result of retracting as assumption. This mechanism must cause all of the justifications that depended, however indirectly, on the retracted assumption to become invalid.

3.7 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.

3.8 Key Words

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

3.9 Self Assessment Questions

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

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.

Paper Code : MCA 402 **Author :Om Parkash**
Lesson No. 04 **Vettor : Saroj**
Lesson Name: Rule Based Systems

Structure

4.0 Objective

The objective of this lesson is to provide an overview of rule-based system. This lesson discuss 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.

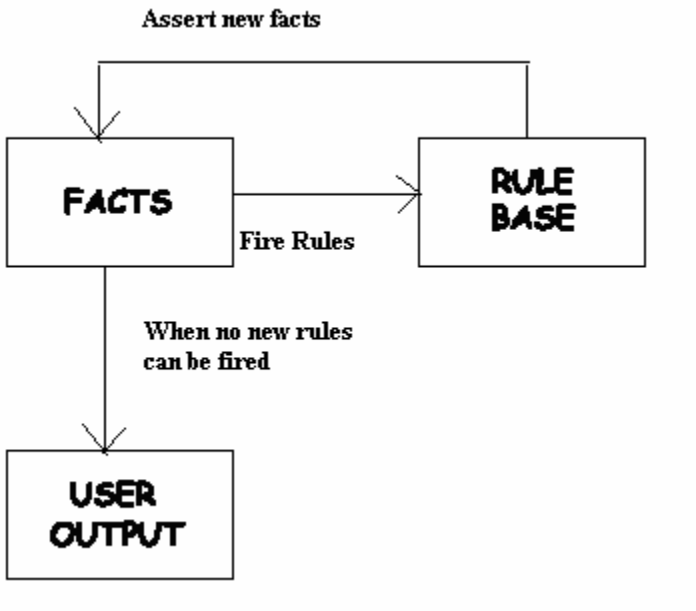
To create a rule-based system for a given problem, you must have (or create) the following:

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

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)→ 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) → 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

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 $\rightarrow$
Square 2 empty and Square 1 contains tile n
Square 1 empty and Square 4 contains tile n $\rightarrow$
Square 4 empty and Square 1 contains tile n
Square 2 empty and Square 1 contains tile n $\rightarrow$
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.

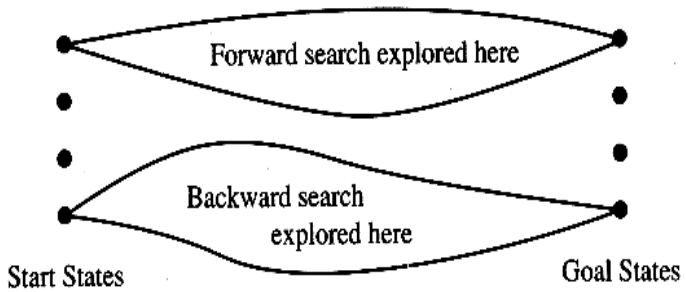


Figure 4.2: A Bad Use of Heuristic Bi-directional Search

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 Key Words

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

4.9 Self Assessments Questions

Answer the following Questions:

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

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.

Paper Code : MCA 402 **Author :Om Parkash**
Lesson No. 05 **Vettor : Saroj**
Lesson Name: Structured Knowledge Representation

Structure

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 pictureized 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 Significance of Knowledge Representation

A representation is a way of describing certain fragments or information so that any reasoning system can easily adopt it for inferencing purpose. Knowledge representation is a study of ways of how knowledge is actually pictureized and how effectively it resembles the representation of knowledge in human brain.

A Knowledge representation system should provide ways of representing complex knowledge and should possess the following characteristics:

- ✓ The representation scheme should have a set of well-defined syntax and semantics.
- ✓ The Knowledge representation scheme should have good expressive capacity. A good expressive capability will catalyze the inferencing mechanism in its reasoning process.
- ✓ From the computer system point of view, the representation must be efficient. By this we mean that it should use only limited resources without compromising on the expressive power.

Major differences between Database and Knowledge Base.

Database	Knowledge Base
Collection of data representing facts.	Has information at higher level of abstraction.
Large volume of data and facts change	Significantly smaller than database and

over time.	changes are gradual.
Operates on a single object.	Operates on a class of objects rather than a single object.
Clerical personnel perform updates.	Domain experts perform updates.
All information needed to be explicitly stated.	Has the power of inferencing.
Maintained for operational purposes.	Used for data analysis and planning.
Represented by relational or network or hierarchical model.	Knowledge representation is by logic or rules or frames or semantic nets.
Predominant way of interaction is by transaction programs and report generators.	Has to have a consultation with the system and provide needed data to obtain the solution.

In this chapter we discuss about some of the widely known representation schemes. They are

1. Semantic Nets
2. Frames
3. Conceptual Dependency
4. Scripts

5.2 Semantics Nets (Associative Network)

A semantic network or a semantic net is a structure for representing knowledge as a pattern of interconnected nodes and arcs. It is also defined as a graphical representation of knowledge. The objects under consideration serve as nodes and the relationships with another nodes give the arcs.

In a semantic net, information is represented as a set of nodes connected to each other by a set of labeled ones, which represent relationships among the nodes. A fragment of a typical semantic net is shown in Figure 5.1.

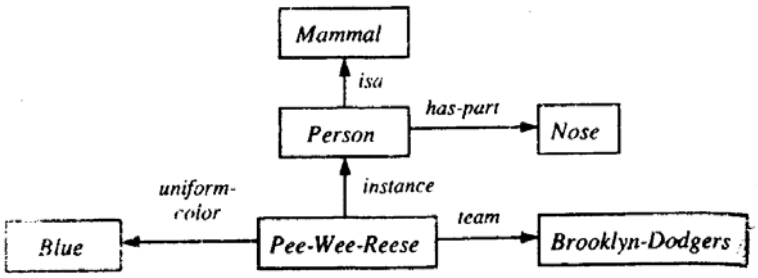


Figure 5.1: A Semantic Network

This network contains example of both the is a and instance relations, as well as some other, more domain-specific relations like team and uniform-color. In this network we would use inheritance to derive the additional relation.
has-part (Pee-Wee-Reese, Nose).

Partitioned Semantic Nets

Suppose we want to represent simple quantified expressions in semantic nets. One was to do this is to partition the semantic net into a hierarchical set of spaces, each of which corresponds to the scope of one or more variables. To see how this works, consider first the simple net shown in Figure 5.2. This net corresponds to the statement.

The dog bit the mail carrier.

The nodes Dogs, Bite, and Mail-Carrier represent the classes of dogs, bitings, and mail carriers, respectively, while the nodes d, b, and m represent a particular dog, a particular biting, and a particular mail carrier. A single net with no partitioning can easily represent this fact.

But now suppose that we want to represent the fact

Every dog has bitten a mail carrier.

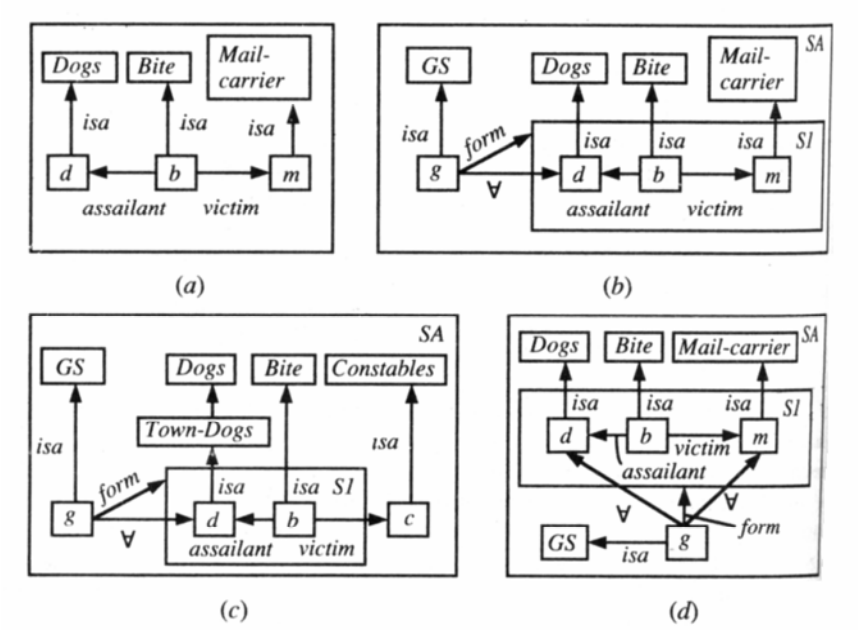


Figure 5.2: Using Partitioned Semantic Nets

$$\forall x : \mathrm{Dog}(x) \rightarrow \exists y : \mathrm{Mail} - \mathrm{Carrier}(y) \wedge \mathrm{Bite}(x, y)$$

It is necessary to encode the scope of the universally quantified variable x in order to represent this variable. This can be done using partitioning as shown in Figure 5.2 (b). The node stands for the assertion given above. Node g is an instance of the special class GS of general statements about the world (i.e., those with universal quantifiers). Every element to GS has at least two attributes: a form, which states the relation that is being assert one or more $\forall$ connections, one for each of the universally quantified variable. In this example, there is only

one such variable d, which can stand for any element the class Dogs. The other two variables in the form, b and m, are understood to existentially qualified. In other words, for every dog d, there exists a biting event and a mail carrier m, such that d is the assailant of b and m is the victim.

To see how partitioning makes variable quantification explicit, consider next similar sentence:

Every dog in town has bitten the constable

The representation of this sentence is shown in Figure 5.2 (c). In this net, the node representing the victim lies outside the form of the general statement. Thus it is not viewed as an existentially quantified variable whose value may depend on the value of d. Instead it is interpreted as standing for a specific entity (in this case, a particular table), just as do other nodes in a standard, no partitioned net. Figure 5.2(d) shows how yet another similar sentences

Every dog has bitten every mail carrier.

should be represented. In this case, g has two $\forall$ links, one pointing to d, which represents dog, and one pointing to m, representing ay mail carrier.

The spaces of a partitioned semantic net are related to each other by an inclusion Search. For example, in Figure 5.2(d), space SI is included in space SA. Whenever search process operates in a partitioned semantic net, it can explore nodes and arcs in space from which it starts and in other spaces that contain the starting point, but it does not go downward, except in special circumstances, such as when a form arc is being traversed. So, returning to figure 5.2(d), from node d it can be determined that d must be a dog. But if we were to start at the node Dogs and search for all known instances dogs by traversing is a likes, we would not find d since it and the link to it are in space SI, which is at a lower level than space SA, which contains Dogs. This is constant, since d does not stand for a particular dog; it is merely a variable that can be initiated with a value that represents a dog.

5.3 Frames

Marvin Minsky in the book on computer vision proposed frames as a means of representing common-sense knowledge. In that Minsky proposed that knowledge is organized into small “packets” called frames. The contents of the frame are certain slots, which have values. All frames of a given situation constitute the system. A frame can be defined as a data structure that has slots for various objects and a collection of frames consists of exceptions for a given situation. A frame structure provides facilities for describing objects, facts about situations, procedures on what to do when a situation is encountered. Because of this facilities a frames are used to represent the two types of knowledge, viz., declarative/factual and procedural.

Default Frames

Set theory provides a good basis for understanding frame systems. Although not all frame systems are defined this way, we do so here. In this view, each frame represents either a class (a set) or an instance (an element of a class). To see how this works, consider the frame system shown in Figure 6.5. In this example, the frames Person, Adult-Male, ML-Baseball-Player (corresponding to major league baseball players) Pitter, and ML-Baseball-Team (for major league baseball team) are all classes. The frame Pee-Wee-Reese and Brooklyn-Dodgers are instances.

The is a relation that we have been using without a precise definition is in fact the subset relation. The isa of adult males is a subset of the set of people. The set of major league baseball players is a subset of the set of adult males, and so forth. Our instance relation corresponds to the relation element of Pee Wee Reese that is an element of the set of fielders. Thus he is also an element of all of the superset of fielders, including major league baseball players and people. The transitivity of isa that we have taken for granted in our description of property inheritance follows directly from the transitivity of the subset relation.

Both the isa and instance relations have inverse attributes, where we call subclasses and all-instances. We do not bother to write them explicitly in our examples unless we need to refer to them. We assume that the frame system maintains them automatically. Either explicitly or by computing them if necessary.

Since a class represents a set, there are two kinds of attributes that can be associated with it. There are attributes about the set itself, and there are attributes that are to be inherited by each element of the set. We indicate the difference between these two by prefixing the latter with an asterisk (*). For example, consider the class ML-Baseball-Player. We have shown only two properties of it as a set: It is a subset of the set of adult males. And it has cardinality 624 (i.e., there are 624 major league baseball players). We have listed five properties that all major league baseball players have (height, bats, batting-average, team, and uniform-colour), and we have specified default values for the first three of them. By providing both kinds of slots, we allow a class both to define a set of objects and to describe a prototypical object of the set.

Sometimes, the distinction between a set and an individual instance may not be seen clearly. For example, the team Brookln-Dodgers, which we have described as a instance of the class of major league baseball teams, could be thought of as a set of players in fact, notice that the value of the slot players is a set. Suppose, instead, what we want to represent the Dodgers as a class instead of an instance. Then its instances would be the individual players. It cannot stay where it is in the isa hierarchy; it cannot be a subclass of ML-Baseball-Team, because if it were, then its elements, namely the players would also, by the transitivity of subclass, be elements of ML-Baseball-team, which is not what we want so say. We have to put it somewhere else in the isa hierarchy. For example, we could make it a subclass of major league baseball players. Then its elements, the players, are also elements of ML-Baseball-Players, Adult-Male, and Person. That

is acceptable. But if we do that, we lose the ability to inherit properties of the Dodges from general information about baseball teams. We can still inherit attributes for the elements of the team, but we cannot inherit properties of the team as a whole, i.e., of the set of players. For example, we might like to know what the default size of the team is, that it has a manager, and so on. The easiest way to allow for this is to go back to the idea of the Dodgers as an instance of ML-Baseball-Team, with the set of players given as a slot value.

Person	
isa:	Mammal
cardinality:	6,000,000,000
*handed:	Right
Adult-Male	
isa:	Person
cardinality:	2,000,000,000
*height;	5-10
ML-Baseball-Player	
isa:	Adult-Male
cardinality:	624
*height:	6-1
*bats:	equal to handed
*batting-average:	. 252
*team:	
*uniform-color:	
Fielder	
Isa:	ML-Baseball-Player
cardinality:	36
*batting-average	.262
Johan	
insance:	Fielder
height:	5-10
bats:	Right
batting-average:	. 309
team:	Brooklyn-Dodgers
uniform-color:	Blue
ML-Baseball-Team	
isa:	Team
cardinality:	26
*team-size:	24
*manager:	24
Brooklyn-Dodgers	
instance:	ML-Baseball-Team

team-size: 24
manager: Leo-Durocher
players: (Johan,Pee-Wee-Reese,…)

Figure 5.3: A Simplified Frame System

But what we have encountered here is an example of a more general problem. A class is a set, and we want to be able to talk about properties that its elements possess. We want to use inheritance to infer those properties from general knowledge about the set. But a class is also an entity in itself. It may possess properties that belong not to the individual instances but rather to the class as a whole. In the case of Brooklyn-Dodgers, such properties included team size and the existence of a manager. We may even want to inherit some of these properties from a more general kind of set. For example, the Dodgers can inherit a default team size from the set of all major league baseball teams. To support this, we need to view a class as two things simultaneously: a subset (isa) of a larger class that also contains its elements and an instance (instance) of a class of sets, from which it inherits its set-level properties.

To make this distinction clear, it is useful to distinguish between regular classes, whose elements are individual entities, and meta classes, which are special classes whose elements are themselves classes. A class is now an element of (instance) some class (or classes) as well as a subclass (isa) of one or more classes. A class inherits properties from the class of which it is an instance, just as any instance does. In addition, a class passes inheritable properties down from is super classes to its instances.

Let's consider an example. Figure 5.4 shows how we would represent teams as classes using this distinction. Figure 5.5 shows a graphic view of the same classes. The most basic met class in the class represents the set of all classes. All classes are instance of it, either directly or through one of its subclasses. In the example, Team in a subclass (subset) of Class and ML-Baseball-Team is a subclass of Team. The class introduces the attribute cardinality, which is to be inherited by all instances of Class (including itself). This makes sense that all the instances of Class are sets and all sets have cardinality.

Team represents a subset of the set of all sets, namely those elements are sets of players on a team. It inherits the property of having cardinality from Class. Team introduces the attribute team-size, which all its elements possess. Notice that team-size is like cardinality in that it measures the size of a set. But it applies to something different cardinality applies to sets of sets and is inherited by all

elements of Class. The slot team-size applies to the element of those sets that happen to be teams. Those elements are of individuals.

ML-Baseball-Team is also an instance of Class, since it is a set. It inherits the property of having cardinality from the set of which it is an instance, namely Class. But it is a subset of Team. All of its instances will have the property of having a team-size since they are also instances of the super class Team. We have added at this level the additional fact that the default team size is 24; so all instance of ML-Baseball-Team will inherit that as well. In addition, we have added the inheritable slot manager.

Brooklyn-Dodgers in an instance of a ML-Baseball-Team. It is not an instance of Class because its elements are individuals, not sets. Brooklyn-Dodgers is a subclass of ML-Baseball-Player since all of its elements are also elements of that set. Since it is an instance of a ML-Baseball-Team, it inherits the properties team-size and manager, as well as their default values. It specifies a new attribute uniform-colour, which is to be inherited by all of its instances (who will be individual players).

Class
instance : Class
isa : Class
*cardinanality :

Team
istance : Class
isa : Class
cardinality : { the number of teams that exist}
* team size : { each team has a size}

ML – Baseball – Team
instance : Class
isa : Team
cardinality : 26 { the number of baseball teams that exist}
* team-size : 24 { default 24 players on a team}
* manager :

Brooklyn-Dodgers
instance : ML – Baseball – Team

isa :	ML-Baseball – Player
team-size :	24
manager :	Leo – Durocher
* unifoirm-color	Blue

Pee-Wee – Reese

instance :	Broklyn – Dodgers
instance :	Fielder
uniform-color:	Blue
batting –average :	309

Figure 5.3 : Representing the Class of All Teams as a Metaclass

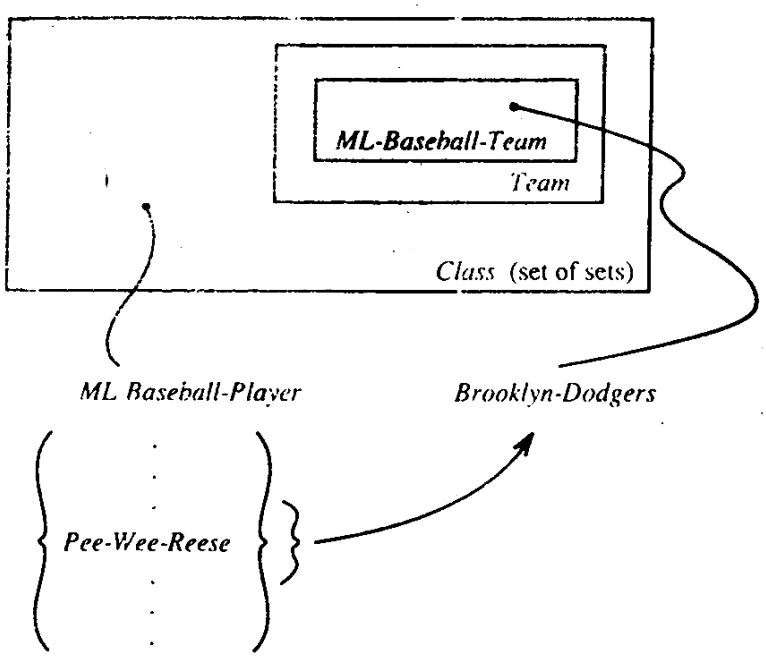


Figure 5.4: Classes and Metaclasses

Finally, Pee-Wee-Reese is an instance of Brooklyn-Dodgers. That makes him also, by transitivity up isa links, an instance of ML-Baseball-Player. But recall that in earlier example we also used the class Fielder, to which we attached the fact that fielders have above average batting averages. To allow that here, we simply make Pee Wee an instance of Fielder as well. He will thus inherit properties from both Brooklyn-Dodgers and from fielder, as well as from the classes above these. We need to guarantee that when multiple inheritances occurs, as it does here, that it works correctly. Specified in this case, we need to assure that batting – average gets inherited from Fielder and not from ML-Baseball-Player through Brooklyn-Dodgers.

In all the frame system we illustrate, all classes are instances of the metaclass Class. As a result, they all have the attribute cardinality out of our description of our examples, though unless there is some particular reason to include them.

Every class is a set. But not every set should be described as a class. A class describes a set of entries that share significant properties. In particular, the default information associated with a class can be used as a basis for inferring values for the properties if it's in individual elements. So there is an advantage to representing as a class these sets for which membership serves as a basis for nonmonotonic inheritance. Typically, these are sets in which membership is not highly ephemeral. Instead, membership is on some fundamental structural or functional properties. To see the difference, consider the following sets:

- People
- People who are major league baseball players
- People who are on my plane to New York

The first two sets can be advantageously represented as classes, with which a sub-statistical number of inheritable attributes can be associated. The last, though, is different. The only properties that all the elements of that set probably share are the definition of the set itself and some other properties that follow from the definition (e.g. they are being transported from one place to another). A simple set, with some associated assertions, is adequate to represent these facts: non-monotonic inheritance is not necessary.

5.4 Scripts

Frames represented a general knowledge representation structure, which can accommodate all kinds of knowledge. Scripts on the other hand, help exclusively in representing stereotype events that takes place in day-to-day activities. Some such events are:

- ✓ Going to hotel, eating something, paying the bill and existing.
- ✓ Going to threatre, getting a ticket, viewing the film/drama and leaving.
- ✓ Going to bank for withdrawal, filling the withdrawal slip/cheque, presenting to the cashier, getting money and leaving the bank.

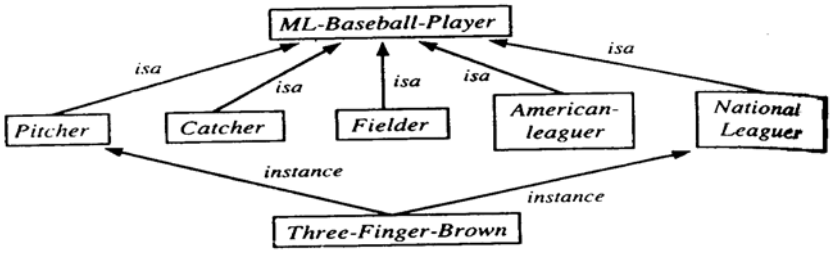
A script is a knowledge representation structure that is extensively used for describing stereotype sequences of actions. It is the special case frame structure. These are intended for capturing situations in which behavior is very stylized. Similar to frames, scripts do have slots and with each slot, we associate information about the slot. Scripts tell people that what can happen in a situation, what events follow and what role every actor plays. It is possible to visualize the same and scripts present a way of representing them effectively what a reasoning mechanism exactly understands what happens at that situation.

5.5 Slots Exceptions

We have provided a way to describe sets of objects and individual objects, both in terms of attributes and values. Thus we have made extensive use of attributes,

which we have represented as slots attached to frames. But it turns out that there are several means why we would like to be able to represent attributes explicitly and describe their properties. Some of the properties we would like to be able to represent and use in meaning include:

- The classes to which the attribute can be attached, i.e. for what classes does it make sense? For example, weight makes sense for physical objects but not for conceptual ones (except in some metaphorical sense).
- A value that all instances of a class must have by the definition of the class.
- A default value for the attribute.
- Rules for inheriting values for the attribute. The usual rule is to inherit down isa and instance links. But some attributes inherit in other ways. For example last-name inherits down the child of link.



ML-Baseball-Player
is-covered by : { Pitcher,Catcher, Fielder}
{ American-Leaguer , National-Leagwer}

Pitcher
isa : ML-Baseball –Player
mutually – disjoint with: { catcher, Fielder}

Catcher
isa : ML-Baseball – Player
mutually-disjoint –with : {Pitcher, Fielder}

Fielder
isa : ML-Baseball Player
mutually –disjoint-with : { Pitcher, Catcher}

American – Leaguer
isa : ML-Baseball –Player
mutually-disjoint-with { National-Leaguer }

National Leaguer
isa : ML-Baseball-Player

mutually-disjoint-with : {american-Leaguer}

Three-Finger-Brown
instance : Pitcher
instance : National – Leaguer

Figure 5.5 : Representing Relationships among Classes

- Rules for computing a value separately from inheritance. One extreme form of such a rule is a procedure written in some procedural programming language such as LISP.
- An inverse attribute.
- Whether the slot is single – valued or multivalued.

In order to be able to represent these attributes of attributes, we need to describe attributes (slots) as frames. These frames will be organized into an isa hierarchy, just as any other frames for attributes of slots. Before we can describe such a hierarchy in detail, we need to formalize our notion of a slot.

A slot is a relation. It maps from elements of its domain (the classes for which it makes sense) to elements of its range (its possible values). A relation is a set of ordered pair. Thus it makes sense to say that one relation (R_1) is a subset of another (R_2). In the case, R_1 is a specification of R_2, so in our terminology is a (R_1, R_2). Since a slot is yet the set of all slots, which we will call Slot, is a metaclass. Its instances are slots, which may have subslots.

Figure 5.5 and 5.6 illustrate several examples of slots represented as frames of slot metaclass. Its instances are slots (each of which is a set of ordered pairs). Associated with the metaclass are attributes that each instance(i.e. each actual slot) will inherit. Each slot, since it is a relation, has a domain and a range. We represent the domain in the slot labelled domain. We break up the representation of the range into two parts: contains logical expressions that further constrain the range to be TRUE. The advantage to breaking the description apart into these two pieces is that type checking a such cheaper than is arbitrary constraint checking, so it is useful to be able to do it separately and early during some reasoning processes.

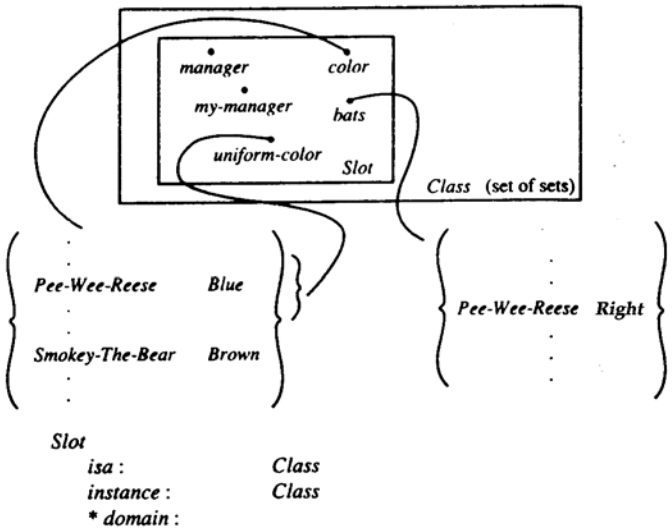
The other slots do what you would expect from their names. If there is a value for definition, it must be propagated to all instances of the slot. If there is a value for default, that value is inherited to all instance of the slot unless there is an overriding value. The attribute transfers lists other slots from which values for this slot can be derived through inheritance. The to-compute slot contains a procedure for deriving its value. Inverse, sometimes they are not useful enough in reasoning to be worth representing. And single valued is used to mark the special cases in which the slot is a function and can have only one value.

Of course, there is a no advantage of representing these properties of slots if there is a reasoning mechanism that exploits them. In the rest of our discussion

we assume for the frame system interpreter knows how to reason with all of these slots of slots as part of its built-in reasoning capability. In particular, we assume that it is capable of forming the following reasoning actions.

- Consistency checking to verify that when a slot value is added to a frame

  - The slot makes sense for the frame. This relies on the domain attribute of the slot.



Slot
isa : Class
instance : Class
domain :
range :
range-constraint :
definition :
default :
trangulars-through :
to-compute :
inverse :
single-valued :

manager
instance : Slot
domain : ML-Baseball – Team
range : Person
range-constraint : kx (baseball-experience x, manager)
default :

inverse : manager – of
single – valued : TRUE

Figure 6. 9 ; Representing Slots as Frames , I

My – manager
instance : Slot
domain : ML-Baseball
Player
range : Person
range-constraint : kx(baseball-
experience x any manager)
to-compute : kx(x,team),
manager
single-valued : TRUE

Colour
instance : Slot
domain : Physical-Object
range : Colour-Set
transfer-through : top-level-part-of
visual-salience : High
single-valued : FALSE

Uniform-colour
instance : Slot
isa : colour
domain : team – Player
range : Colour – Set
range-constraint : non-Pink
visual-sailence : High
single-valued : FALSE

Bats
instance : Slot
domain : ML-Baseball-
Player
range : (Left,Right,
Switch)
to-compute : kx x, handed
single-valued : TRUE

Figure 5.6 : Representing Slots as Frames II

- The value is a legal value for the slot. This relies on the range and range – constraint attributes.

- Maintenance of consistency between the values for slots and their inverses whenever one is updated.

- Propagation of definition values along isa and instance links.
- Inheritance of default values along isa and instance links.
- Computation of a value of a slot as needed. This relies on the to-compute and transfers through attributes.
- Checking that only a single value is asserted for single –valued slots. Replacing an old value by the new one when it is asserted usually does this. An alternative is to force explicit retraction of the old value and to signal a contradiction if a new value is asserted when another is already there.

There is something slightly counterintuitive about this way of defining slots. We have defined properties range – constraint and default as parts of a slot. But we then think of them as being properties of a slot associated with a particular class. For example in Figure 5.7, we listed two defaults for the batting – average slot, one associated with major league baseball players and one associated with fielders. Figure 5.6 shows how this can be represented correctly by creating a specialization of batting average that can he associated with a specialization of ML-Baseball-Player to represent the more special information that is known about the specialized class. This seems cumbersome. It is natural, though given our definition of a slot as relation. There are really two relations here, one a specialization of the other. And below we will define inheritance so that it looks for values of either the slot it is given or any of that slot's generations.

Unfortunately, although this model of slots is simple and it is internally consisted it is not easy to see. So we introduce some notational shorthand that allows the four most important properties of a slot (domain range definition and default) to be defined implicitly by how the slot is used in the definitions of the classes in its domain. We describe the domain implicitly to be the class where the slot appears. We describe the range and any range constraints with the clause MUST BE, as the value of an inherited slot. Figure 5.8 shows an example of this notation. And we describe the definition and the default. If they are present by inserting them as the value of the slot when one appears. The two will be distinguished by perplexing a definitional value with an assts (“). We then let the underlying book keeping of the frame system create the frames to represent slots as they are needed.

Now let's look at examples of how these slots can be used. The slots bats and my manager illustrate the use of the to-compute attribute of a slot. The variable x will be bound to the frame to which the slot is attached. We are the notation to specify the value of a slot of a frame. Specially, x, y describes the value (s) of the y slot it frame x. So we know that to compute a frame a value for any manager, it is necessary find the frame's value for team, then find the resulting team's manager. We have simply composed two slots to form a new one. Computing the value of the bats slots is a even simpler. Just go get the value of the hand slot.

Batting average

instance :	Slot
domain :	ML-Baseball Player
range :	Number

range-constraint : kx(0 < x range-constraint < 1)
default : 252
single-valued : TRUE

Fielder batting average
instance : Slot
isa : batting-average
domain : Fielder
range : Number
range-constraint : kx 9o < x,range – constraint < 1)
default : 262
single-valued : TRUE

Figure 5.7 Associating Defaults with Slots

ML-Baseball-Player

Bats : MUST BE (Left, Right, Switch)

Figure 5.8. A Shorthand Notation For Slot – Range Specification

The manager slots illustrate the use of a range constraint. It is stated in terms of a variable x, which is bound to the frame whose manager slot is being described. It requires that any manager be not only a person but someone with baseball experience relies on the domain-specific function baseball experience, which must be defined somewhere in the system.

The slots colour and uniform–colour illustrate the arrangements of slots in is history. The relation colour is a fairly general one that holds between physical objects colour. The attribute uniform-colour is a restricted form of colour that applies only to team players and ones that are allowed for team uniform (anything but pink). Arranging slots in a hierarchy is useful for the same reason than arranging any thing else in a hierarchy is, it supports inheritance. In this example the general slot is known to have high visual salience. The more specific slot uniform colour then tests this property, so it too is known to have high visual salience.

The slot colour also illustrates the use of the transfer-through slot, which defines a way of computing a slot's value by retrieving it from the same slot of a related object as its example. We used transfers through to capture the fact that if you take an object and chop it up into several top level parts (in other words, parts that are not contained for each other) then they will all be the same colour. For example, the arm of a sofa is the colour as the sofa. Formally what transfers through means in this example is

John

Height : 72

Bill

Height :

Figure 5.9 Representing Slot-Values

color(x,y) $\wedge$ top-level–part-of(z,x)$\rightarrow$ color(z,y)
In addition to these domain independent slot attributes slots may have domain specific properties that support problem solving in a particular domain. Since the frame system interpreter does not treat these slots explicitly, they will be useful precisely to the extent that the domain problem solver exploits them.

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 Self Assessments Questions

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

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.

Paper Code : MCA 402 **Author :Om Parkash**
Lesson No. 06 **Vettor : Saroj**
Lesson Name: Handling Uncertainty and Learning

Structure

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 Probabilistic Reasoning

Here we will examine methods that use probabilistic representations for all knowledge and which reason by propagating the uncertainties from evidence and assertions to conclusions. As before, the uncertainties can arise from an inability to predict outcomes due to unreliable, vague, incomplete, or inconsistent knowledge.

The probability of an uncertain event A is a measure of the degree of likelihood of occurrence of that event. The set of all possible events is called the sample space; S A probability measure is a function P(A) which maps event outcome $E_1, E_2, ...$from S into real numbers and which satisfies the following axioms of probability:

1. for any event $A \subseteq S.$
2. $P(S)$=1, a certain outcome
3. For $E_i \cap E_j = \Phi$, for all $i \neq j$ (the E_i are mutually exclusive), $P(E_i \cup E_2 \cup E_3 \cup ...) = P(E_2) + P(E_3) + ...$

From these three axioms and the rules of set theory, the basic law of probability can be derived. Of course, the axioms are not sufficient to compute the

probability of an outcome. That requires an understanding of the underlying distributions that must be established through one of the following approaches:

1. Use of a theoretical argument that accurately characterizes the processes.
2. Using one's familiarity and understanding of the basic processes to assign subjective probabilities, or
3. Collecting experimental data from which statistical estimates of the underlying distributions can be made.

Since much of the knowledge we deal with is uncertain in nature, a number of our beliefs must be tenuous. Our conclusions are often based on available evidence and past experience, which is often far from complete. The conclusions are, therefore, no more than educated guesses. In a great many situations it is possible to obtain only partial knowledge concerning the possible outcome of some event. But, given that knowledge, one's ability to predict the outcome is certainly better than with no knowledge at all. We manage quite well in drawing plausible conclusions from incomplete knowledge and past experiences.

Probabilistic reasoning is sometimes used when outcomes are unpredictable. For example, when a physician examines a patient, the patient's history, symptoms, and test results provide some, but not conclusive, evidence of possible ailments. This knowledge, together with the physician's experience with previous patients, improves the likelihood of predicting the unknown (disease) event, but there is still much uncertainty in most diagnoses. Likewise, weather forecasters "guess" at tomorrow's weather based on available evidence such as temperature, humidity, barometric pressure, and cloud coverage observations. The physical relationships that overrun these phenomena are not fully understood; so predictability is far from certain. Even a business manager must make decisions based on uncertain predictions when the market for a new product is considered. Many interacting factors influence the market, including the target consumer's lifestyle, population growth, potential consumer income, the general economic climate, and many other dependent factors.

In all of the above cases, the level of confidence placed in the hypothesized conclusions is dependent on the availability of reliable knowledge and the experience of the human prognosticator. Our objective in this chapter is to describe some approaches taken in AI systems to deal with reasoning under similar types of uncertain conditions.

6.2 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.

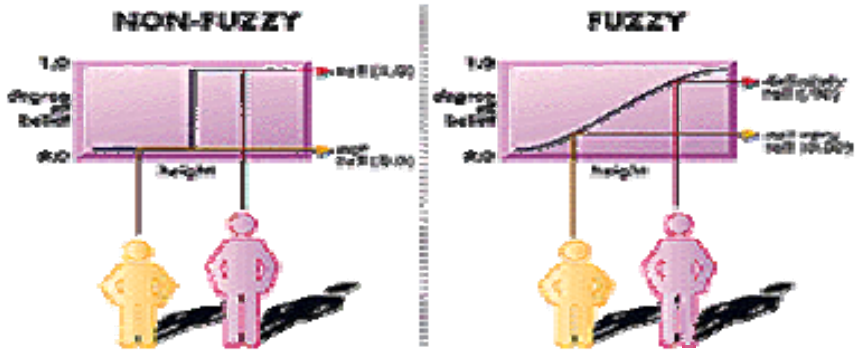
The two measures MB and MD are combined into a single measure called the certainty factor (CF), defined by

$$CF(H,E) = MB(H,E) - MD(H,E) \qquad (6.13)$$

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.



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?

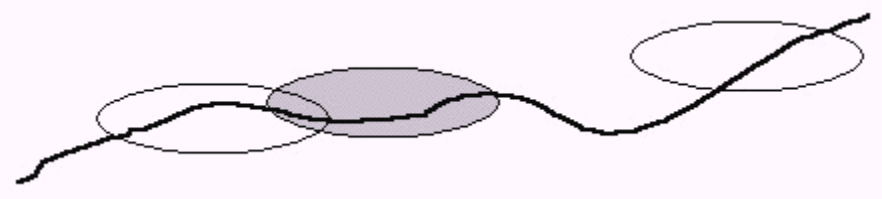
The first applications of fuzzy theory were primaly industrial, such as process control for cement kilns. However, as the technology was further embraced, fuzzy logic was used in more useful applications. In 1987, the first fuzzy logic-controlled subway was opened in Sendai in northern Japan. Here, fuzzy-logic 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.



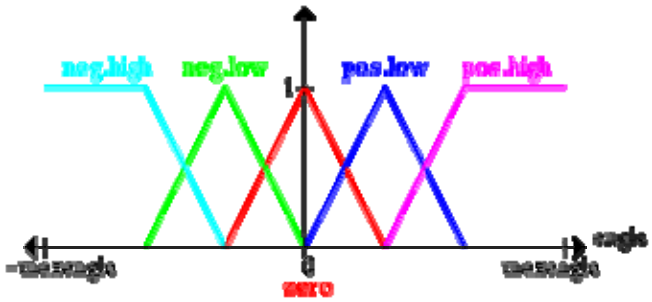
Fuzzy Patches
In a fuzzy system this simply means that all our rules can be seen as patches and the input and output of the machine can be associated together using these patches. Graphically, if the *rule patches* shrink, our fuzzy subset triangles gets narrower. Simple enough? Yes, because even novices 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:

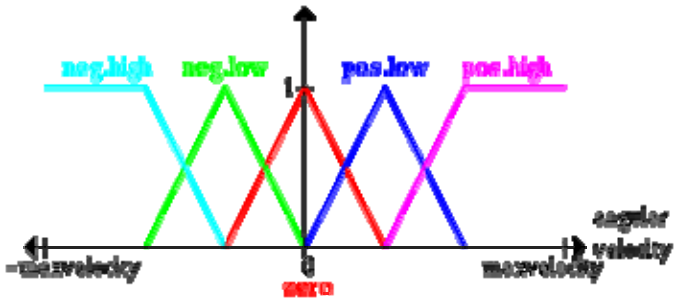
1) Fuzzification (Using membership functions to graphically describe a situation)
2) Rule evaluation (Application of fuzzy rules)
3) Defuzzification (Obtaining the crisp or actual results)
As a simple example on how fuzzy controls are constructed, consider the following classic situation: the inverted 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

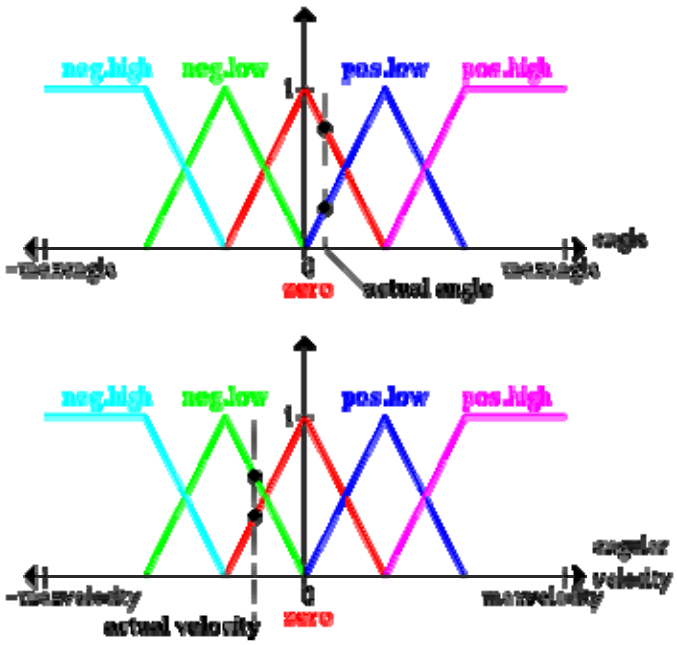


Note: For simplicity, it is assumed that all membership functions are spreaded equally. Hence, this explains why no actual scale is included in the graphs.

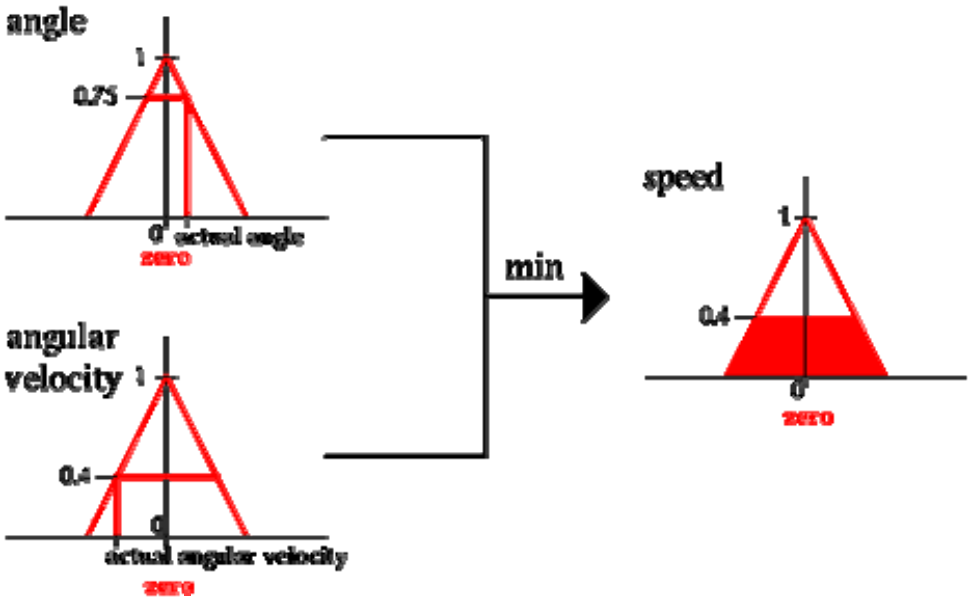
Step 2

The next step is to define the fuzzy rules. The fuzzy rules are mearly 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 summarised in the table below. The dashes are for conditions, which have no rules ascociated with them. This is don eto simplify the situation.

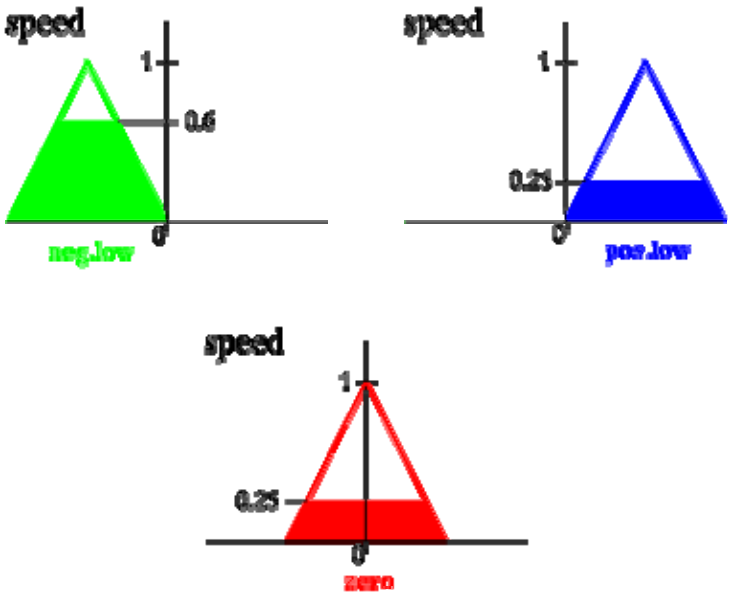
An application of these rules is shown using specific values for angle and angular velocities. The values used for this example are 0.75 and 0.25 for zero and positive-low angles, and 0.4 and 0.6 for zero and negative-low angular velocities. These points sre on the graphs below.



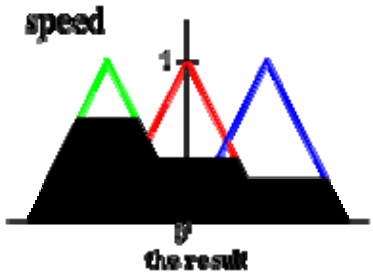
Consider the rule "if angle is zero and angular velocity is zero, the speed is zero". The actual value belongs to the fuzzy set zero to a degree of 0.75 for "angle" and 0.4 for "angular velocity". Since this is an AND operation, the minimum criterion is used , and the fuzzy set zero of the variable "speed" 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".

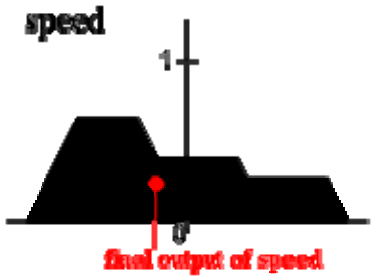


The four results overlaps and is reduced to the following figure



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.

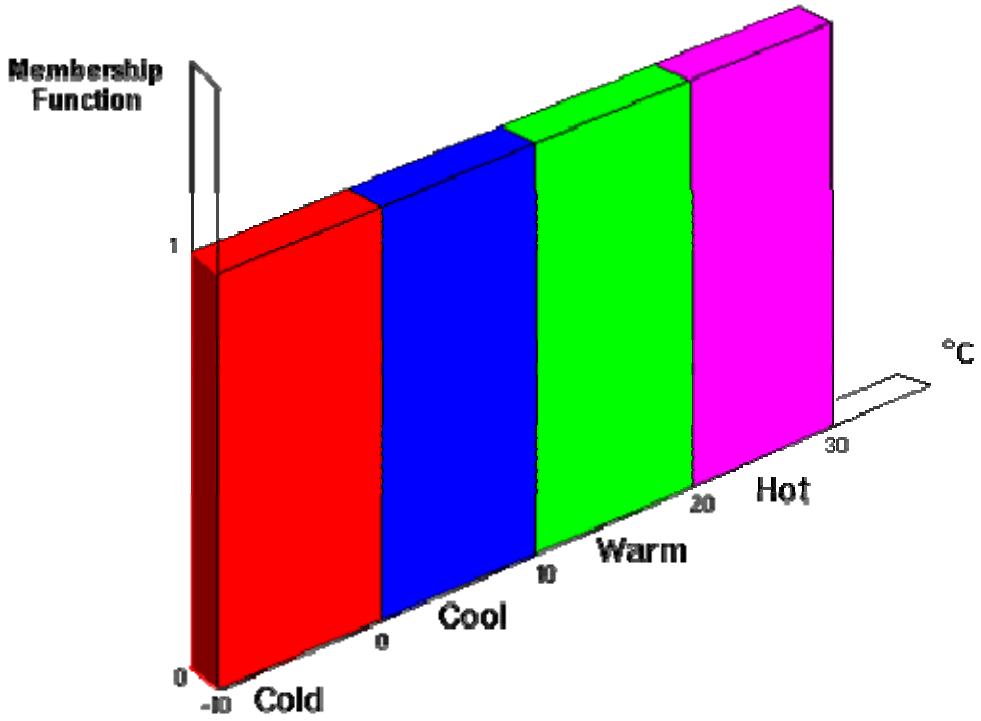


Fig. 1 : Bivalent Sets to Characterize the Temp. of a room.

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.

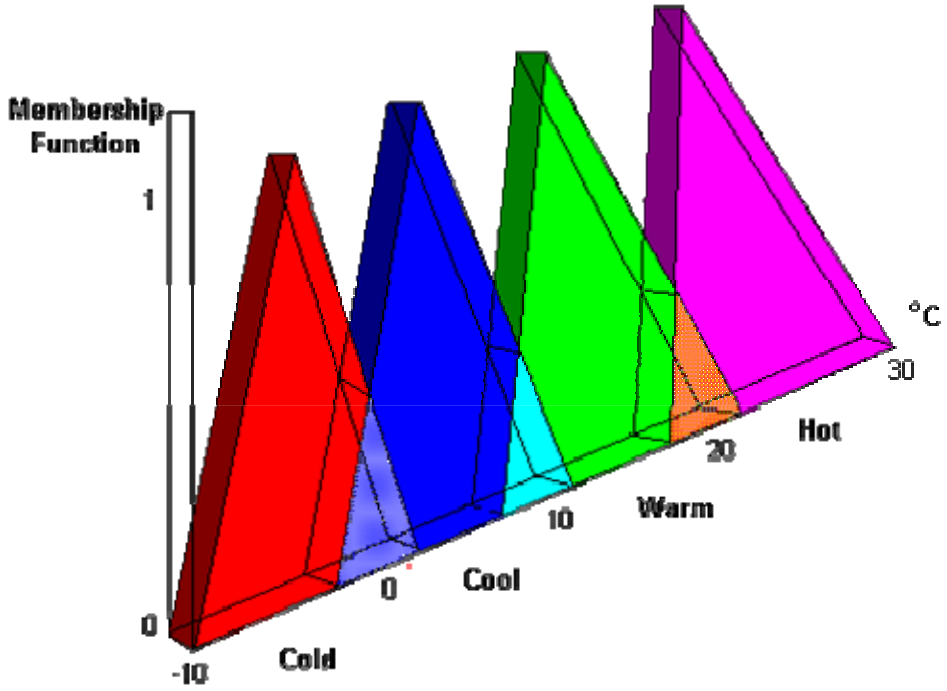


Fig. 2 - Fuzzy Sets to characterize the Temp. of a room.

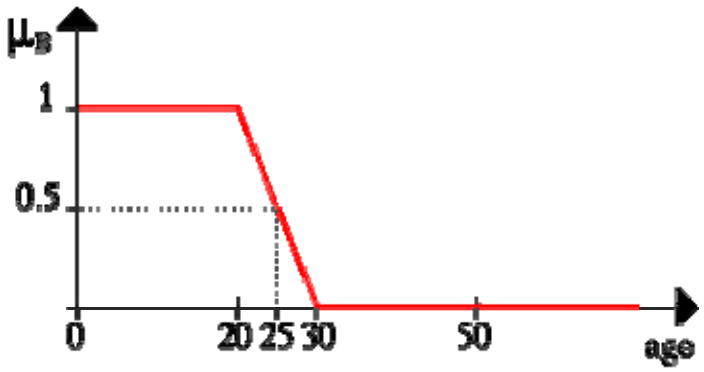
The whole concept can be illustrated with this example. Let's talk about people and "youthness". In this case the set S (the universe of discourse) is the set of people. A fuzzy subset YOUNG is also defined, which answers the question "to what degree is person x young?" To each person in the universe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

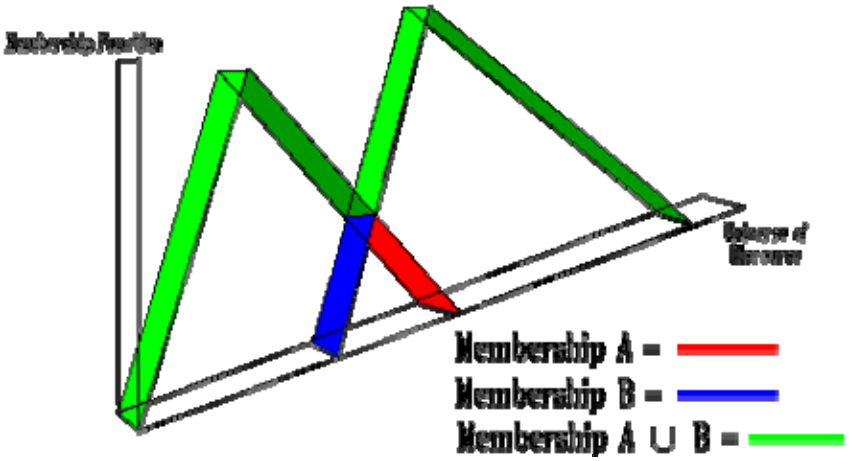
Note: Membership functions almost never have as simple a shape as age(x). They will at least tend to be triangles pointing up, and they can be much more complex than that. Furthermore, membership functions so far is discussed as if they always are based on a single criterion, but this isn't always the 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.

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

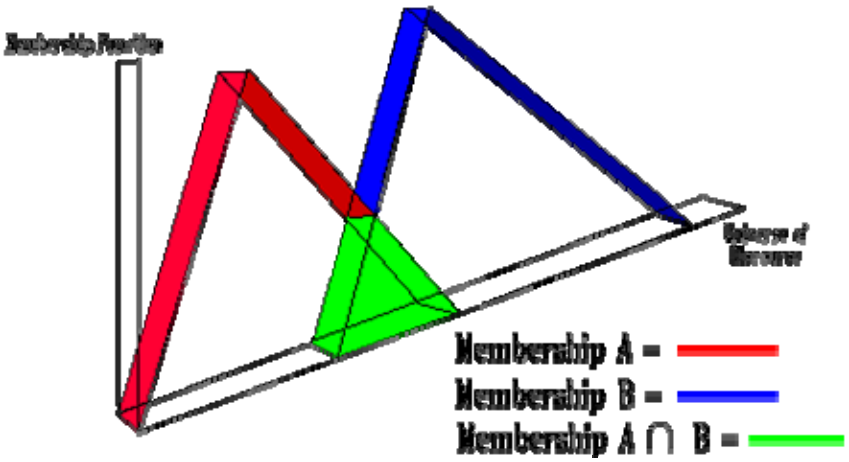


The Union operation in Fuzzy set theory is the equivalent of the **OR** operation in Boolean algebra.

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.

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

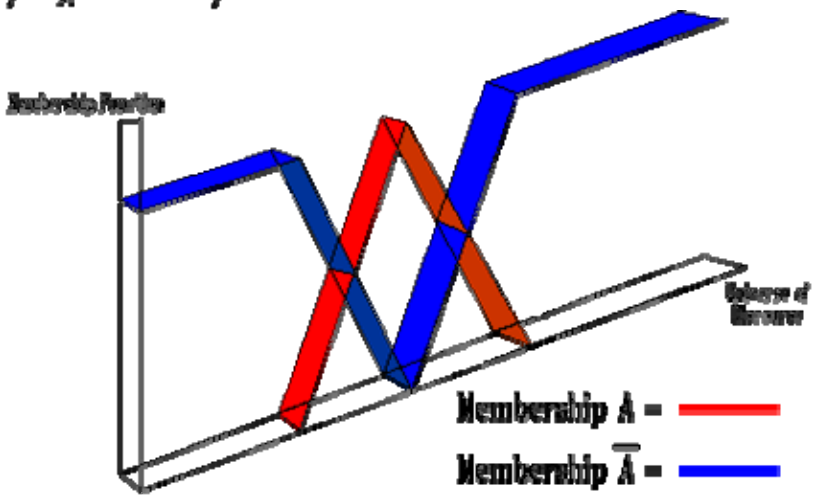


The Intersection operation in Fuzzy set theory is the equivalent of the **AND** operation in Boolean algebra.

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.

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



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

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

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

Glossary

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.

6.4 Concept of Learning

One of the most often heard criticisms of AI is that machines cannot be called intelligent until they are able to learn to do new things and to adapt to new situations, rather than simply doing as they are told to do. There can be little question that the ability to adapt to new surroundings and to solve new problems is an important characteristic of intelligent entities. Can we expect to see such abilities in programs? Ada Augusta, one of the earliest philosophers of computing, wrote that

The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform.

Several AI critics have interpreted this remark as saying that computers cannot learn. In fact, it does not say that at all. Nothing prevents us from telling a computer how to interpret its inputs in such a way that its performance gradually improves.

Rather than asking in advance whether it is possible for computers to "learn," it is much more enlightening to try to describe exactly what activities we mean when we say "learning" and what mechanisms could be used to enable us to perform those activities. Simon has proposed that learning denotes changes in the system that are adaptive in the sense that they enable the system to do the same task or tasks drawn from the same population more efficiently and more effectively the next time.

As thus defined, learning covers a wide range of phenomena. At one end of the spectrum is skill refinement. People get better at many tasks simply by practicing. The more you ride a bicycle or play tennis, the better you get. At the other end of the spectrum lies knowledge acquisition. As we have seen, many AI programs draw heavily on knowledge as their source of power. Knowledge is generally acquired through experience and such acquisition is the focus of this chapter.

Knowledge acquisition itself includes many different activities. Simple storing of computed information, or rote learning, is the most basic learning activity. Many computer programs, e.g., database systems, can be said to "learn" in this sense,

although most people would not call such simple storage, learning. However, many AI programs are able to improve their performance substantially through rote-learning technique and we will look at one example in depth, the checker-playing program of Samuel.

Another way we learn is through taking advice from others. Advice taking is similar to rote learning, but high-level advice may not be in a form simple enough for a program to use directly in problem solving. The advice may need to be first operationalized.

People also learn through their own problem-solving experience. After solving a Complex problem, we remember the structure of the problem and the methods we used to solve it. The next time we see the problem, we can solve it more efficiently. Moreover, we can generalize from our experience to solve related problems more easily contrast to advice taking, learning from problem-solving experience does not usually involve gathering new knowledge that was previously unavailable to the learning program. That is, the program remembers its experiences and generalizes from them, but does not add to the transitive closure of its knowledge, in the sense that an advice-taking program would, i.e., by receiving stimuli from the outside world. In large problem spaces, however, efficiency gains are critical. Practically speaking, learning can mean the difference between solving a problem rapidly and not solving it at all. In addition, programs that learn though problem-solving experience may be able to come up with qualitatively better solutions in the future.

Another form of learning that does involve stimuli from the outside is learning from examples. We often learn to classify things in the world without being given explicit rules. For example, adults can differentiate between cats and dogs, but small children often cannot. Somewhere along the line, we induce a method for telling cats from dogs - based on seeing numerous examples of each. Learning from examples usually involves a teacher who helps us classify things by correcting us when we are wrong. Sometimes, however, a program can discover things without the aid of a teacher.

AI researchers have proposed many mechanisms for doing the kinds of learning described above. In this chapter, we discuss several of them. But keep in mind throughout this discussion that learning is itself a problem-solving process. In fact, it is very difficult to formulate a precise definition of learning that distinguishes it from other problem-solving tasks.

The five different learning methods are as follows

1. Memorization (rote learning)

   Learning by memorization is the simplest form of learning. It requires the least amount of inference and is accomplished by simply copying the knowledge in the same form that it will be used directly into the knowledge base. We use this type of learning when we memorize

2. Direct Instruction (by being told)

It is slightly different more complex form of learning. This type of learning requires more inference than rote learning since the knowledge must be transformed into an operational form before being integrated into the knowledge base. We use this type of learning when a teacher presents a number of facts directly to us in well-organized manner.

3. Analogy

Analog learning is the process of learning a new concept or solution through the use of similar known concepts or solutions. We use this type of learning, when solving problems on an exam where previously learned examples serve as a guide or when we learn to drive a truck using our knowledge of car. This form of learning requires still more inferring than either of the previous forms, since difficult transformations must be made between the known and unknown situations.

4. Induction

It is the power full form of learning which, like analogical learning, also requires the use of inferring than the first two methods. This form of learning requires the use inductive inference, a form of invalid but useful inference. We use inductive learning when we formulate a general concept after seeing a number of instances or examples of the concept.

5. Deduction

It is accomplished through a sequence of deductive inference steps using known facts. Fro known facts, new facts or relationship

6.5 Learning Automation

The theory of learning automata was first introduced in 1961 (Tsetlin, 1961). Since that time these systems have been studied intensely, both analytically and through simulations (Lakshmivarahan, 1981). Learning automata systems are finite set adaptive systems, which interact iteratively with a general environment. Through a probabilistic trial-and-error response process they learn to choose or adapt to a behavior that produces the best response. They are, essentially, a form of weak, inductive learners.

From Figure given below, we see that the learning model for learning automata has been simplified for just two components, an automaton (learner) and an environment. The learning cycle begins with an input to the learning automata system from the environment. This input elicits one of a finite number of possible responses and then provides some form of feedback to the automaton in return. The automaton to alter its stimulus-response mapping structure to improve its behavior in a more favorable way uses this feedback.

As a simple example, suppose a learning automata is being used to learn the best temperature control setting for your office each morning. It may select any

one of ten temperature range settings at the beginning of each day. Without any prior knowledge of your temperature preferences, the automaton randomly selects a first setting using the probability vector corresponding to the temperature settings.

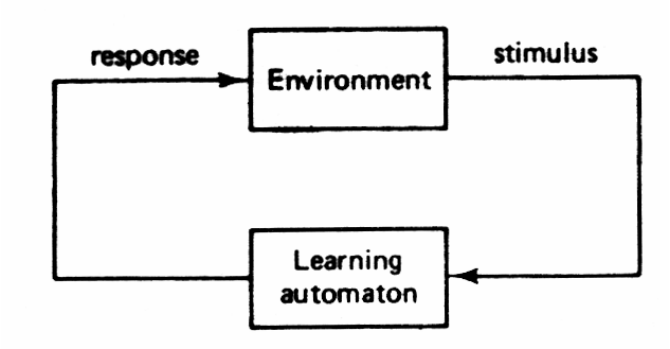


Figure Learning Automaton Model

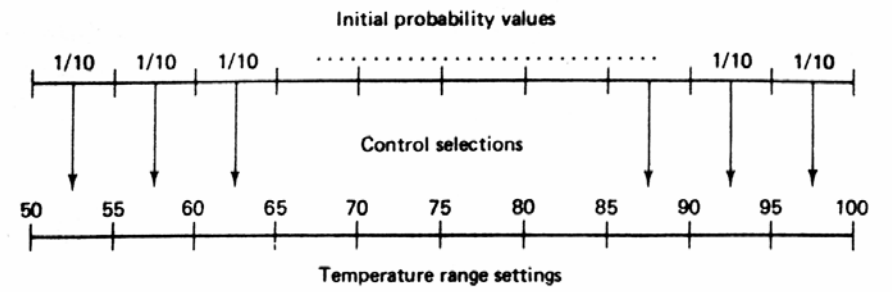


Figure: Temperature Control Model

Since the probability values are uniformly distributed, any one of the settings will be selected with equal likelihood. After the selected temperature has stabilized, the environment may respond with a simple good-bad feedback response. If the response is good, the automata will modify its probability vector by rewarding the probability corresponding to the good setting with a positive increment and reducing all other probabilities proportionately to maintain the sum equal to 1. If the response is bad, the automaton will penalize the selected setting by reducing the probability corresponding to the bad setting and increasing all other values proportionately. This process is repeated each day until the good selections have high probability values and all bad choices have values near zero. Thereafter, the system will always choose the good settings. If, at some point, in the future your temperature preferences change, the automaton can easily readapt.

Learning automata have been generalized and studied in various ways. One such generalization has been given the special name of collective learning automata (CLA). CLAs are standard learning automata systems except that feedback is not provided to the automaton after each response. In this case, several collective stimulus-response actions occur before feedback is passed to the automaton. It has been argued (Bock, 1976) that this type of learning more closely resembles that of human beings in that we usually perform a number or group of primitive actions before receiving feedback on the performance of such

actions, such as solving a complete problem on a test or parking a car. We illustrate the operation of CLAs with an example of learning to play the game of Nim in an optimal way.

Nim is a two-person zero-sum game in which the players alternate in removing tokens from an array that initially has nine tokens. The tokens are arranged into three rows with one token in the first row, three in the second row, and five in the third row (Figure 7.10).

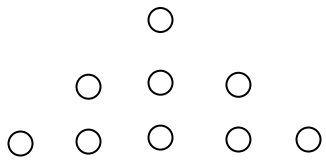


Figure : Nim Initial Configuration

The first player must remove at least one token but not more than all the tokens in any single row. Tokens can only be removed from a single row during each payer's move. The second player responds by removing one or more tokens remaining in any row. Players alternate in this way until all tokens have been removed; the loser is the player forced to remove the last token.

We will use the triple (n_1, n_2, n_3) to represent the states of the game at a given time where n_1, n_2 and n_3 are the numbers of tokens in rows 1, 2, and 3, respectively. We will also use a matrix to determine the moves made by the CLA for any given state. The matrix of Figure 7.11 has heading columns which correspond to the state of the game when it is the CLA's turn to move, and row headings which correspond to the new game state after the CLA's turn to move, and row headings which correspond to the new game state after the CLA has completed a move. Fractional entries in the matrix are transition probabilities used by the CLA to execute each of its moves. Asterrisks in the matrix represent invalid moves.

Beginning with the initial state (1, 3, 5), suppose the CLA's opponent removes two tokens from the third row resulting in the new state (1, 3, 3). If the ClA then removes all three tokens from the second row, the resultant state is (1, 0, 3). Suppose the opponent now removes all remaining tokens from the third row. This leaves the CLA with a losing configuration of (1, 0, 0).

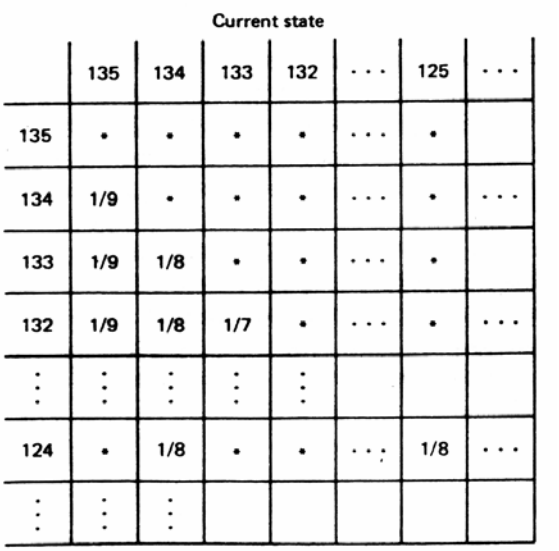


Figure: CLA Internal Representation of Game States

As the start of the learning sequence, the matrix is initialized such that the elements in each column are equal (uniform) probability values. For example, since there are eight valid moves from the state (1, 3, 4) each column element under this state corresponding to a valid move has been given uniform probability values corresponding to all valid moves for the given column state.

The CLA selects moves probabilistically using the probability values in each column. So, for example, if the CLA had the first move, any row intersecting with the first column not containing an asterisk would be chosen with probability $\frac{1}{9}$.

This choice then determines the new game state from which the opponent must select a move. The opponent might have a similar matrix to record game states and choose moves. A complete game is played before the CLA is given any feedback, at which time it is informed whether or not its responses were good or bad. This is the collective feature of the CLA.

If the CLA wins a game, increasing the probability value in each column corresponding to the winning move rewards all moves made by the CLA during that game. All non-winning probabilities in those columns are reduced equally to keep the sum in each column equal to 1. If the CLA loses a game, reducing the probability values corresponding to each losing move penalizes the moves leading to that loss. All other probabilities in the columns having a losing move are increased equally to keep the column totals equal to 1.

After a number of games have been played by the CLA, the matrix elements that correspond to repeated wins will increase toward one, while all other elements in the column will decrease toward zero. Consequently, the CLA will choose the winning moves more frequently and thereby improve its performance.

Simulated games between a CLA and various types of opponents have been performed and the results plotted (Bock, 1985). It was shown, for example, that

two CLAs playing against each other required about 300 games before each learned to play optimally. Note, however, that convergence to optimality can be accomplished with fewer games if the opponent always plays optimally (or poorly), since, in such a case, the CLA will repeatedly lose (win) and quickly reduce (increase) the losing (winning) move elements to zero (one). It is also possible to speed up the learning process through the use of other techniques such as learned heuristics.

Learning systems based on the learning automaton or CLA paradigm are fairly general for applications in which a suitable state representation scheme can be found. They are also quite robust learners. In fact, it has been shown that an LA will converge to an optimal distribution under fairly general conditions if the feedback is accurate with probability greater 0.5 (Narendra and Thathachar, 1974). Of course, the rate of convergence is strongly dependent on the reliability of the feedback.

Learning automata are not very efficient learners as was noted in the game-playing example above. They are, however, relatively easy to implement, provided the number of states is not too large. When the number of states becomes large, the amount of storage and the computation required to update the transition matrix becomes excessive.

Potential applications for learning automata include adaptive telephone routing and control. Such applications have been studied using simulation programs (Narendra et al., 1977).

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 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.

6.7 Learning by Induction

Classification is the process of assigning, to a particular input, the name of a class to which it belongs. The classes from which the classification procedure can choose can be described in a variety of ways. Their definition will depend on the use to which they will be put.

Classification is an important component of many problem-solving tasks. In its simplest form, it is presented as a straightforward recognition task. An example of this is the question "What letter of the alphabet is this?" But often classification is embedded inside another operation. To see how this can happen, consider a problem-solving system that contains the following production rule:

> If: the current goal is to get from place A to place B, and
>
> there is a WALL separating the two places
>
> then: look for a DOORWAY in the WALL and go through it.

To use this rule successfully, the system's matching routine must be able to identify an object as a wall. Without this, the rule can never be invoked. Then, to apply the rule, the system must be able to recognize a doorway.

Before classification can be done, the classes it will use must be defined. This can be done in a variety of ways, including:

Isolate a set of features that are relevant to the task domain. Define each class by a weighted sum of values of these features. Each class is then defined by a scoring function that looks very similar to the scoring functions often used in other situations, such as game playing. Such a function has the form.

$$C1t_1 + C2V_2 + C3t_3 + ...$$

Each t corresponds to a value of a relevant parameter, and each c represents the weight to be attached to the corresponding t. Negative weights can be used to indicate features whose presence usually constitutes negative evidence for a given class.

For example, if the task is weather prediction, the parameters can be such measurements as rainfall and location of cold fronts. Different functions can be written to combine these parameters to predict sunny, cloudy, rainy, or snowy weather.

Isolate a set of features that are relevant to the task domain. Define each class as a structure composed of those features. For example, if the task is to identify animals, the body of each type of animal can be stored as a structure, with various features representing such things as color, length of neck, and feathers.

There are advantages and disadvantages to each of these general approaches. The statistical approach taken by the first scheme presented here is often more efficient than the structural approach taken by the second. But the second is more flexible and more extensible.

Regardless of the way that classes are to be described, it is often difficult to construct, by hand, good class definitions. This is particularly true in domains that are not well understood or that change rapidly. Thus the idea of producing a classification program that can evolve its own class definitions is appealing. This task of constructing class definitions is called concept learning, or induction. The techniques used for this task J must, of course, depend on the way that classes (concepts) are described. If classes are described by scoring functions, then concept learning can be done using the technique of coefficient adjustment. If, however, we want to define classes structurally, some other technique for learning class definitions is necessary. In this section, we present three such techniques.

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

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

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 Key Words

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 Self Assessments Questions

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

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.

Paper Code : MCA 402 **Author :Om Parkash**
Lesson No. 07 **Vettor : Saroj**
Lesson Name: Expert Systems

Structure

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 What is Expert System?

An Expert System contains knowledge about a specific field to assist human experts or provide information to people who do not have access to an expert in the particular field. An Expert System act as intelligent assistants to human experts. Knowledge Engineer and Domain Expert are the key personnel, work together to design an expert system

7.1.1 Need and Justification of Expert Systems

Human experts in any field are frequently in great demand and are therefore, usually in short supply. One solution of this problem is Expert system. An Expert system may be defined as an AI computer program specially designed to represent human expertise in a particular domain (area of Expertise). Expert systems have been proven to be effective in a number of problem domains, which normally require the kind of intelligence possessed by a human expert.

According to Paul Harmon and David King, expert system can help meet the following needs:

- New approaches to business and productivity,

- Expertise,
- Knowledge
- Competence, and
- Smart automated equipment.

The areas of application are almost endless. Wherever human expertise is needed to solve a problem, expert systems are most likely of the options sought. Application domain includes Medical, law, chemistry, biology engineering, finance, banking, manufacturing, aerospace military operations, meteorology, geology, geophysics and many more. In this lesson we attempts to demystify expert systems by examining, in detail, what they are and how they are developed. Also, case study of MYCIN & RI is provided to increase your familiarities with these remarkable programs.

7.2 Components of an Expert System

Although components of an Expert System vary in their design, most Expert Systems have a knowledge base, an inference engine and a user interface.

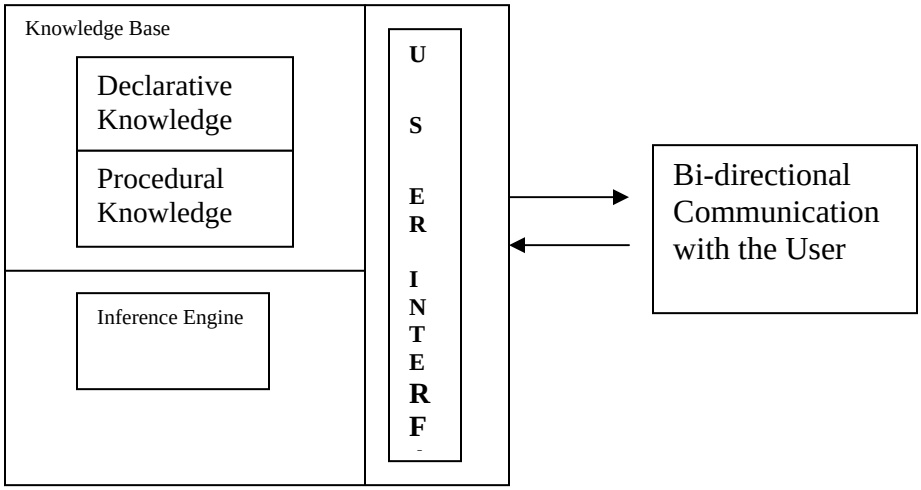


Figure 7.1: - Expert System Components

The component of expert system that contains system's knowledge is called its knowledge base. A knowledge base contains both declarative knowledge (facts about objects, events, and situations) and procedural knowledge (information about course of action). The inference engine of an expert system controls how and when the information to the Knowledge base is applied. The user interface component enables you to communicate with an expert system. The communication performed by a user interface is bi-directional.

7.3 Characteristics Features of an Expert System

Although each system is unique, certain features are desirable for any expert system.

- ✓ The program should be useful.
- ✓ An expert system should be developed to meet a specified need.
- ✓ The program should be usable. An expert system should be designed so that even a layman finds it's easy to use.
- ✓ The program should be educational when appropriate.
- ✓ The program should be able to explain its advice.
- ✓ The program should be respond to simple questions.
- ✓ The program should be learn new knowledge.
- ✓ The program's knowledge should be easily modified.

7.4 Knowledge Acquisition

Knowledge acquisition is the process of adding new knowledge to a knowledge base & refining or otherwise improving knowledge that was previously acquired. Knowledge acquisition is the most important aspect of the expert system development. It is referred to as the process of getting and transforming appropriate information out of an expert's head, document or any source into some manageable form. The idea of getting knowledge from an expert and presenting that information is a very common occurrence eg: Reporters, Journals and writers are a regular conductor of these processes. They are classed as knowledge engineers who interview many people and then publish their information in the newspapers.

The three major approaches for knowledge acquisition are:

1. Interviewing Expert

This involves the knowledge engineer having a face-to-face interview with the expert. This technique does not require any equipment. It's just a verbal talking with each other. Therefore it is important that the knowledge engineer has good communication skills and the expert should be able to express his knowledge with the engineer. The engineer collects lot of information by asking many questions and programs it into the knowledge base.

2. Learning by Being Told

In this approach the expert system user-interface conducts a conversation or discussion with the expert and the expert has to represent and refine his/her own knowledge from what he understands. The knowledge engineer handles the design and makes the activity easier to understand.

3. Learning by Observation

This approach the expert system gives the expert some sample problem outline or case studies, which the expert has to solve. The problems are usually examples of previous events which the expert has to solve using an algorithm known as Induction. This algorithm helps expert to gain knowledge and it simplifies those examples into rules.

7.4.1 Knowledge Acquisition Process

The expert system development uses a methodology known as Rapid Prototyping. These involve selection and development of a section of a system, which is tested on part of the system for refinement, and further development. Once the initial development i.e. the design and knowledge base decisions have been made; a prototype (a trial model) is developed to allow other developer to test their ideas of design. This will enable them to test each stage as it is developed and see if the system is working properly.

7.4.2 Knowledge Acquisition stages

Knowledge acquisition has five stages throughout the development. The stages are as following:

- ✓ ***Identification***

This stage identifies the problems and the knowledge engineer becomes aware of the domain, its goals and selects the correct material.

- ✓ ***Conceptualization***

This defines how the concepts or ideas and the associations between them are outlined and how experts relate them.

- ✓ ***Formalization***

Here the knowledge engineer organizes the concepts, tasks and other information into formal and clear representation.

- ✓ ***Implementation***

Here the knowledge rules are put into a structured form for the expert system tool and a prototype (trial model) is created for testing out the design and the processes. The knowledge engineer has to produce a written documentation that will connect the knowledge base topics with the original data that were created earlier.

- ✓ ***Testing***

The prototype system is tested for its efficiency and accuracy to see if it is working as required. In order to do this a small scenario or problem set is tested and the results from this system are used to alter or improve the prototype system.

7.5 MYCIN

MYCIN is an expert system that helps diagnose bacteriological blood infections. The development of MYCIN began at Stanford University. MYCIN is an expert

system, which diagnoses infectious blood diseases and determines a recommended list of therapies for the patient. As part of the Heuristic Programming Project at Stanford, several projects directly related to MYCIN were also completed including a knowledge acquisition component called THEIRESIUS, a tutorial component called GUIDON, and a shell component called EMYCIN (for Essential MYCIN). EMYCIN was used to build other diagnostic systems including PUFF, a diagnostic expert for pulmonary diseases. EMYCIN also became the design model for several commercial expert system building tools.

MYCIN's performance improved significantly over a period of several year as additional knowledge was added. Tests indicate that MYCIN' performance now equals or exceeds that of experienced physicians. The initial MYCIN knowledge base contained about only 200 rules. This number was gradually increased to more than 600 rules by the early 1980s. The added rules significantly improved MYCIN's performance leading to a 65% success record that compared favorably with experienced physicians who demonstrated only an average 60% success rate.

Subgoaling in MYCIN

MYCIN is a heterogeneous program, consisting of many different modules. There is a part of MYCIN's control structure that performs a quasi-diagnostic function. But the goals to be achieved are not physical goals, involving the movement of objects in space, but reasoning goals that involve the establishment of diagnostic hypothesis.

This section concentrates upon the diagnostic module of MYCIN, giving a simplified account of its function, structure and runtime behavior.

Treating blood infections

Firstly, we need to give a brief description of MYCIN's domain: treatment of blood infections. This description pre-supposes no specialized medical knowledge on the part of the reader. But, as with any expert system, having some understanding of the domain is crucial to understand what the program does.

An 'anti-microbial agent' is any drug designed to kill bacteria or arrest their growth. Some agents are too toxic for therapeutic purposes, and there is no single agent effective against all bacteria. The selection of therapy for bacterial infection can be viewed as a four-part decision process:

- Deciding if the patient has a significant infection;
- Determining the (possible) organism(s) involved;
- Selecting a set of drugs that might be appropriate;
- Choosing the most appropriate drug or combination of drugs.

Samples taken from the site of infection are sent to a microbiology laboratory for culture, that is, an attempt to grow organisms from the sample in a suitable medium.

Early evidence of growth may allow a report of the morphological or staining characteristics of the organism. However, even if an organism is identified, the range of drugs it is sensitive to may be unknown or uncertain.

MYCIN is often described as a diagnostic program, but this is not so. Its purpose is to assist a physician who is not an expert in the field of antibiotics with the treatment of blood infections. In doing so, it develops diagnostic hypotheses and weights them, but it need not necessarily choose between them. Work on MYCIN began in 1972 as collaboration between the medical and AI communities at Stanford University. The most complete single account of this work is Short-life (1976).

There have been a number of extensions, revisions and abstractions of MYCIN since 1976, but the basic version has five components shown in the fig. 7.2, which shows the basic pattern of information flow between the modules.

(1} A-knowledgebase, which contains factual and judgmental knowledge about the domain.

(2) A dynamic patient database containing information about a particular case.

(3) A consultation program, which asks questions, draws conclusions, and gives advice about a particular case based on the patient data and the static knowledge.

(4) An explanation program, which answers questions and justifies its advice, using static knowledge and a trace of the program's execution.

(5) A knowledge acquisition program for adding new rules and changing existing ones.

The system consisting of components (l)-(3) is the problem solving pan of MYCIN, which generates hypotheses with respect to the offending organisms, and makes therapy recommendations based on these hypotheses.

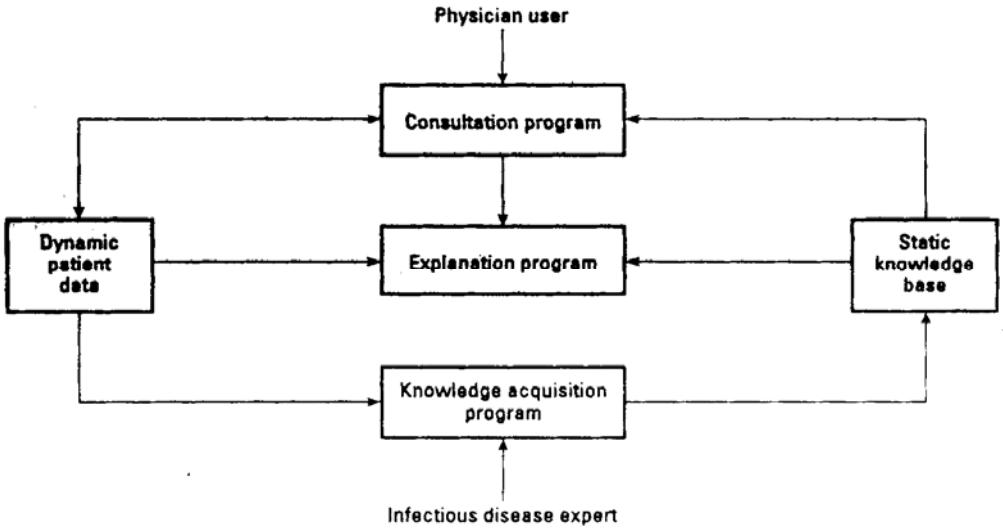


Figure 7.2: Organization of MYCIN

MYCIN's knowledge base

MYCIN's knowledge base is organized around a set of rules of the general form

if condition$_1$ and ... and condition$_m$ hold

then draw conclusion$_1$ and... and conclusion$_n$

encoded as data structures of the LISP programming language

Figure 7.3 shows the English translation of a typical MYCIN rule for inferring class of an organism. The program itself provides this translation. Such rules are called ORGRULES and they attempt to cover such organisms as streptococcus, pseudomonas, and entero-bacteria.

The rule says that if an isolated organism appears rod-shaped, stains in a certain way, and grows in the presence of oxygen, then it is more likely to be in the class entero-bacteria. The number 0.8 is called the tally of the rule, which says how certain conclusion is given, that the conditions are satisfied. The use of the tally is explained below. Each rule of this kind can be thought of as encoding a piece of human knowledge whose applicability depends only upon the context established by the conditions of the rule.

The conditions of a rule can also be satisfied with varying degrees of certainty, the import of such rules roughly is as follows:

if condition$_1$ holds with certainty x_1 ... and condition$_m$ holds with certainty x_m

then draw conclusion$_1$ with certainty y_1 and... and conclusion$_n$ with certainty y_n

where the certainty associated with each conclusion is a function of the combined certainties of the conditions and the tally, which is meant to reflect our degree of confidence in the application of the rule.

In summary, a rule is a premise-action pair and such rules are sometimes called ‘productions' for purely historical reasons. Premises are conjunction of conditions, and their certainty is a function of the certainty of these conditions. Conditions are either proposition, which evaluate the truth or falsehood with some degree of certainty, (for example 'the organism is rod-shaped') or disjunctions of such conditions. Actions are either conclusions to be drawn with some appropriate degree of certainty, for example the identity of some organism, or instructions to be carried out, for example compiling a list of therapies.

We will explore the details of how rules are interpreted and scheduled for application in the following sections, but first we must look at MYCIN's other structures for representing medical knowledge.

IF 1) The stain of the organism is gramneg, and

2) The morphology of the organism is rod, and

3) The aerobicity of the organism is aerobic

THEN There is strongly suggestive evidence (.8) that

the class of the organism is entero-bacteria

A MYCIN ORGRULE for drawing the conclusion enterobacteriaaceae

In addition to rules, the knowledge base also stores facts and definitions in various forms:

- simple lists, for example the list of all organisms known to the system;
- knowledge tables, which contain records of certain clinical parameters and the values they take under various circumstances, for example the morphology (structural shape) of every bacterium known to the system;
- a classification system for clinical parameters according to the context in which they apply, for example whether they are attributes of patients or organisms.

Much of the knowledge not contained in the rules resides in the properties associated with the 65 clinical parameters known to MYCIN. For example, shape is an attribute of organisms which can take on various values, such as 'rod' and 'coccus.' Parameters are also assigned properties by the system for its own purposes. The main ones either (i) help to monitor the interaction with the user, or (ii) provide indexes which guides the application of rules.

Patient information is stored in a structure called the context tree, which serves to organize case data. Figure on next page shows a context tree representing a

particular patient, PATIENT-1, with three associated cultures (samples, such as blood samples, from which organisms may be isolated) and a recent operative procedure that may need to be taken into account (for example, because drugs were involved, or because the procedure involves particular risks of infection). Associated with cultures are organisms that are suggested by laboratory data, and associated with organisms are drugs that are effective against them.

Imagine that we have the following data stored in a record structure associated with the node for ORGANISM-1:

GRAM = (GRAMNEG 1.0)

MORPH = (ROD .8) (COCCUS .2)

AIR = (AEROBIC .6)

with the following meaning:

- the Gram stain of ORGANISM-1 is definitely Gram negative;
- ORGANISM-1 has a rod morphology with certainty 0.8 and a coccus morphology with certainty 0.2;
- ORGANISM-1 is aerobic (grows in air) with certainty 0.6.

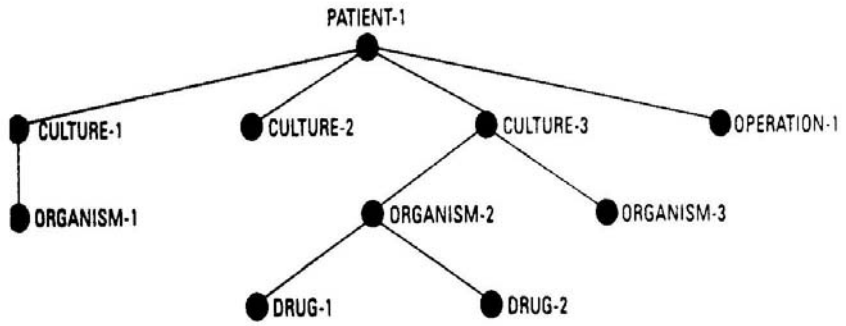


Figure 7.3: A typical MYCIN context tree

Suppose now that the rule of conclusion above is applied. We want to compute the certainty that all three conditions of the rule

IF 1) the stain of the organism is gramneg, and

2) the morphology of the organism is rod, and

3) the aerobicity of the organism is aerobic

THEN there is strongly suggestive evidence (0.8) that the class of the organism is entero-bacteria.

are satisfied by the data. The certainty of the individual conditions is 1.0, 0.8 and 0.6 respectively, and the certainty of their conjunction is taken to be the minimum of their individual certainties, hence 0.6.

The idea behind taking the minimum is that we are only confident in a conjunction of conditions to the extent that we are confident in its least inspiring element. This is rather like saying that a chain is only as strong as its weakest link. By an inverse argument, we argue that our confidence in a disjunction of conditions is as strong as the strongest alternative, that is, we take the maximum. This convention forms part of a style of inexact reasoning called fuzzy logic.

In the case, we draw the conclusion that the class of the organism is entero-bacteria with a degree of certainty equal to

$$0.6 \times 0.8 = 0.48$$

The 0.6 represents our degree of certainty in the conjoined conditions, while the 0.8 stands for our degree of certainty in the rule application. These degrees of certainty are called certainty factors (CFs). Thus, in the general case,

$$CF(action) \times CF(premise) \times CF(rule).$$

Where we revisit the whole topic of how to represent uncertainty. It turns out that the CF model is not always in agreement with the theory of probability; in other words, it is not always correct from a mathematical point of view. However, the computation of certainty factors is much more tractable than the computation of the right probabilities, and the deviation does not appear to be very great in the MYCIN application.

MYCIN's control structure

MYCIN has a top-level goal rule which define the whole task of the consultation system, which is paraphrased below:

IF 1) there is an organism which requires therapy and

2) consideration has been given to any other organisms requiring therapy

THEN compile a list of possible therapies, and determine the best one in this list.

A consultation session follows a simple two-step procedure:

• Create the patient context as the top node in the context tree;

• Attempt to apply the goal rule to this patient context.

Applying the rule involves evaluating its premise, which involves finding out if there is indeed an organism, which requires therapy. In order to find this out, it

must first find out if there is indeed an organism present which is associated with a significant disease. This information can either be obtained from the user directly, or via some chain of inference based on symptoms and laboratory data provided by the user.

The consultation is essentially a search through a tree of goals. The top goal at the root of the tree is the action part of the goal rule, that is, the recommendation of a drug therapy. Subgoals further down the tree include determining the organism involved and seeing if it is significant. Many of these subgoals have subgoals of their own, such as finding out the stain properties and morphology of an organism. The leaves of the tree are fact goals, such as laboratory data, which cannot be deduced.

A special kind of structure, called an AND/OR tree, is very useful for representing the way in which goals can be expanded into subgoals by a program. The basic idea is that root node of the tree represents the main goal, terminal nodes represent primitive actions that can be carried out, while non-terminal nodes represent subgoals that are susceptible to further analysis. There is a simple correspondence between this kind of analysis and the analysis of rule sets.

Consider the following set of condition-action rules:

if X has BADGE and X has GUN, then X is POLICE

if X has REVOI.VER or X as PISTOL or X has RIFLE, then X has GUN

if X has SHIELD, then X has BADGE

We can represent this rule set in terms of a tree of goals, so long as we maintain the distinction between conjunctions and disjunctions of subgoals. Thus, we draw an arc between the links connecting the nodes BADGE and GUN with the node POLICE, to signify that both subgoals BADGE and GUN must be satisfied in order to satisfy the goal POLICE. However, there is no arc between the links connecting REVOLVER and PISTOL and RIFLE with GUN, because satisfying either of these will satisfy GUN. Subgoals as BADGE can have a single child, SHIELD, signifying that a shield counts as a badge.

The AND/OR tree in Figure 7.4 can be thought of as a way of representing the search space for POLICE, by enumerating the ways in which different operators can be applied in order to establish POLICE as true.

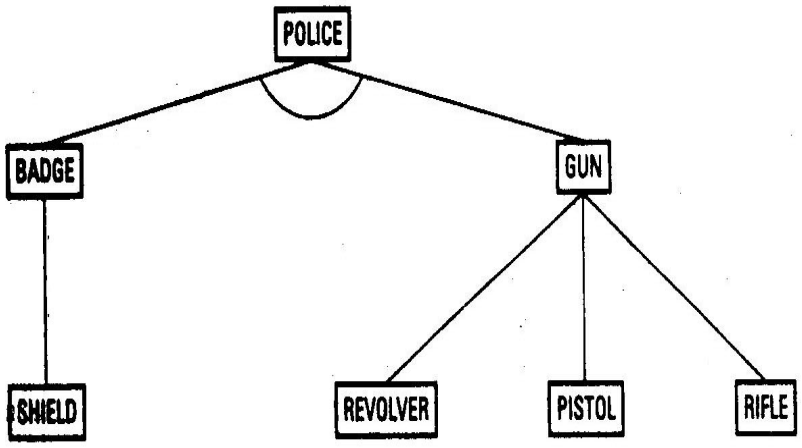


Figure 7.4: Representing a rule set as an AND/OR tree

This kind of control structure is called backward chaining, since the program reasons backward from what it wants to prove towards the facts that it needs, rather than reasoning forward from the facts that it possesses. In MYCIN, goals were achieved by breaking them down into sub goals to which operators could be applied. Searching for a solution by backward reasoning is generally more focused than forward chaining, as we saw earlier, since one only considers potentially relevant facts.

MYCIN's control structure uses an AND/OR tree, and is quite simple as AI programs go;

(1) Each sub goal set up is always a generalized form of the original goal. So, if the sub goal is to prove the proposition that the identity of the organism is E. Coli, then the subgoal actually set up is to determine the identity of the organism. This initiates an exhaustive search on a given topic, which collects all of the available evidence about organisms.

(2) Every rule relevant to the goal is used, unless one of them succeeds with certainty. If more than one rule suggest a conclusion about a parameter, such as the nature of the organism, then their results are combined. If the evidence about a hypothesis falls between -0.2 and +0.2, it is regarded as inconclusive, and the answer is treated as unknown.

(3) If the current subgoal is a leaf node, then attempt to satisfy the goal by asking the user for data. Else set up the subgoal for further inference, and go to (1).

The selection of therapy takes place after this diagnostic process has run its course. It consists of two phases: selecting candidate drugs, and then choosing a preferred drug, or combination of drugs, from this list.

Evidence Combination

In MYCIN, two or more rules might draw conclusions about a parameter with different Weights of evidence. Thus one rule might conclude that the organism is E. Coli with a certainty of 0.8, while another might conclude from other data that it is E. Coli with a certainty of 0.5 or – 0.8. In the case of a certainty less than zero, the evidence is actually against the hypothesis.

Let X and Y be the weights derived from the application of different rules. MYCIN combines these weights using the following formula to yield the single certainty factor.

$$CF(X, Y) = \begin{cases} X + Y - XY & X, Y > 0 \\ X + Y + XY & X, Y < 0 \\ (X + Y)/(1 - \min(|X|, |Y|)) & \text{otherwise} \end{cases}$$

where |X| denotes the absolute value of X.

One can see what is happening on an intuitive basis. If the two pieces of evidence both confirm (or disconfirm) the hypothesis, then confidence in the hypothesis goes up (or down). If the two pieces of evidence are in conflict, then the denominator dampens the effect.

This formula can be applied more than once, if several rules draw conclusions about the same parameter. It is commutative, so it does not matter in what order weights are combined.

IF the identity of the organism is pseudomonas

THEN I recommend therapy from among the following drugs:

1 CCLISTIN (.98)

2 POLYMYXIN (.96)

3 QENTAMICIN (.96)

4 CARBENICILLIN (.65)

5 SULFISOXAZOLE (.64)

A MYCIN therapy rule

The special goal rule at the top of the AND/OR tree does not lead to a conclusion, but instigates actions, assuming that the conditions in the premise are satisfied. At this point, MYCIN's therapy rules for selecting drug treatments come into play; they contain sensitivities information for the various organisms known to the system. A sample therapy rule is given above.

The numbers associated with the drug are the probabilities that a pseudomonas will be sensitive to the indicated drug according to medical statistics. The

preferred drug is selected from the list according to criteria, which attempts to screen for contra-indications of the drug and minimize the number of drugs administered, in addition to maximizing sensitivity. The user can go on asking for alternative therapies until MYCIN runs out of options, so the pronouncements of the program are not definitive.

7.6 RI

RI (sometimes also called XCON) is a program that configures DEC VAX systems. Its rules look like this:

> If: The most current active context is distributing massbus devices, and
>
> There is a single-port disk drive that has not been' assigned to a massbus, and
>
> The number of devices that each massbus should support is known, and
>
> There is a massbus that has been assigned at least
>
> One disk drive and that should support additional disk drives and
>
> The type of cable needed to connect the disk drive to the previous device on the massbus is known
>
> then
>
> Assign the disk drive to the massbus.

Notice that RI's rules, unlike MYCIN's, contain no numeric measures of certainty. In the task domain with which RI deals, it is possible to state exactly the correct thing to be done in each particular set of circumstances (although it may require a relatively complex set of antecedents to do so). One reason for this is that there exists a good deal of human expertise in this area. Another is that since RI is doing a design task (in contrast to the diagnosis task performed by MYCIN), it is not necessary to consider all possible alternatives; one good one is enough. As a result, probabilistic information is not necessary in RI.

PROSPECTOR is a program that provides advice on mineral exploration. Its rules look like this:

> If: Magnetite or pyrite in disseminated or vein let form is present
>
> then (2, -4) there is favourable mineralization and texture for the propylitic stage.

In PROSPECTOR, each rule contains two confidence estimates. The first indicates the extent to which the presence of the evidence described in the condition part of the rule suggests the validity of the rule's conclusion. In the

PROSPECTOR rule shown above, the number 2 indicates that the presence of the evidence is mildly encouraging. The second-confidence estimate measures the extent to which the evidence is necessary to the validity of the conclusion, or stated another way, the extent to which the lack of the evidence indicates that the conclusion is not valid. In the example rule shown above, the number -4 indicates that the absence of the evidence is strongly discouraging for the conclusion.

DESIGN ADVISOR is a system that critiques chip designs. Its rules look like:

If	The sequential 'level count of ELEMENT is greater than 2, UNLESS the signal of ELEMENT is resetable
then	Critique for poor resetability
DEFEAT	Poor resetability of ELEMENT
due to	Sequential level count of ELEMENT greater than 2
by	ELEMENT is directly resetable

The DESIGN ADVISOR gives advice to a chip designer, who can accept or reject the advice. If the advice is rejected, then system can exploit a justification-based truth maintenance system to revise its model of the circuit. The first rule shown here says that an element should be criticized for poor resetability if its sequential level count is greater than two, unless its signal is currently believed to be resetable. Resetability is a fairly common condition, so it is mentioned explicitly in this first rule. But there is also a much less common condition, called direct resetability. The DESIGN ADVISOR does not even bother to consider that condition unless it gets in trouble with its advice. At that point, it can exploit the second of the rules shown above. Specifically, if the chip designer rejects a critique about resetability and if that critique was based on a high level count, then the system will attempt to discover (possibly by asking the designer) whether the element is directly resetable. If it is, then the original rule is defeated and the conclusion withdrawn.

Reasoning with the Knowledge

As these example rules have shown, expert systems exploit many of the representation and reasoning mechanisms that we have discussed. Because these programs are usually, written primarily as rule-based systems, forward chaining, backward chaining, or some combination of the two, is usually used. For example, MYCIN used backward chaining to discover what organisms were present; then it used forward chaining to reason from the organisms to a treatment regime. RI, on the other hand, used forward chaining. As the field of expert systems matures, more systems that exploit other kinds of reasoning mechanisms are being developed. The DESIGN ADVISOR is an example of such a system; in addition to exploiting rules, it makes extensive use of a justification-based truth maintenance system.

Expert System Shells

Initially, each expert system that was built was created from scratch, usually in LISP. But, after several systems had been built this way, it became clear that

these systems often had a lot in common. In particular, since the systems were constructed as a set of declarative representations (mostly rules) combined with an interpreter for those representations, it was possible to separate the interpreter from the domain-specific knowledge and thus to create a system that could be used to construct new expert systems by adding new knowledge corresponding to the new problem domain. The resulting interpreters are called shells. One influential example of such a shell is EMYCIN (for Empty MYCIN), which was derived from MYCIN.

There are now several commercially available shells that serve as the basis for many of the expert systems currently being built. These shells provide much greater flexibility in representing knowledge and in reasoning with it than MYCIN did. They typically support rules, frames, truth maintenance systems, and a variety of other reasoning mechanisms.

Early expert system shells provided mechanisms for knowledge representation, reasoning, and explanation. Later, tools for knowledge acquisition were added. Expert system shells needed to do something else as well. They needed to make it easy to integrate expert systems with other kinds of programs. Expert systems cannot operate in a vacuum, any more than their human counterparts can. They need access to corporate databases, and access to them needs to be controlled just as it does for other systems. They are often embedded within larger application programs that use primarily conventional programming techniques. So one of the important features that a shell must provide is an easy-to-use interface between an expert system that is written with the shell and a larger, probably more conventional, programming environment.

7.7 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

7.8 Key words

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

7.9 Self Assessment Questions (SAQ)

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.

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.