Rajalakshmi Institute of Technology
Department of Information Technology

CS6659 - Artificial Intelligence

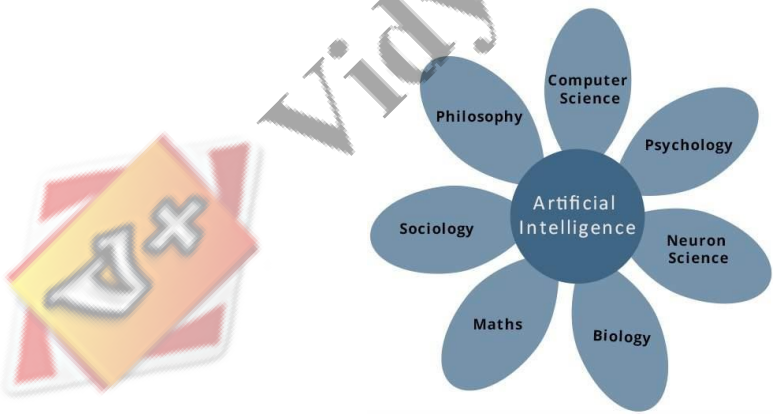
Unit-1 INTRODUCTION TO AI AND PRODUCTION SYSTEMS

Introduction to AI-Problem formulation, Problem Definition - Production systems, Control strategies, Search strategies. Problem characteristics, Production system characteristics - Specialized productions system- Problem solving methods – Problem graphs, Matching, Indexing and Heuristic functions -Hill Climbing-Depth first and Breath first, Constraints satisfaction – Related algorithms, Measure of performance and analysis of search algorithms.

1. Introduction to Artificial Intelligence

Definition: ***Artificial Intelligence (AI) is a branch of Science which deals with helping machines to find solutions to complex problems in a more human-like fashion.***

- This generally involves borrowing characteristics from human intelligence, and applying them as algorithms in a computer friendly way.
- AI is the branch of computer science that attempts to approximate the results of human reasoning by organizing and manipulating factual and heuristic knowledge.
- AI is generally associated with *Computer Science*, but it has many important links with other fields such as *Maths*, *Psychology*, *Cognition*, *Biology* and *Philosophy*, among many others.



- Areas of AI activity include expert systems, natural language understanding, speech recognition, vision, and robotics.
- Our ability to combine knowledge from all these fields will ultimately benefit our progress in the quest of creating an intelligent artificial being.
- The term was coined in 1956 by John McCarthy at the Massachusetts Institute of Technology.
- There are several programming languages that are known as AI languages because they are used almost exclusively for AI applications. The two most common are **LISP** (List Processing) and **Prolog** (Programming Logic)**.**

- The greatest advances have occurred in the field of games playing. The best computer chess programs are now capable of beating humans. In May, 1997, an IBM super-computer called Deep Blue defeated world chess champion Gary Kasparov in a chess match.

1.1 Branches of Artificial intelligence

- **Games playing**: programming computers to play games such as chess and checkers.
- **Expert systems**: programming computers to make decisions in real-life situations (for example, some expert systems help doctors diagnose diseases based on symptoms).
- **Natural Language Processing**: programming computers to understand natural human languages.
- **Neural Networks**: Systems that simulate intelligence by attempting to reproduce the types of physical connections that occur in human brains.
- **Robotics**: programming computers to see and hear and react to other sensory stimuli currently, no computers exhibit full artificial intelligence (that is, are able to simulate human behavior).

1.2 Applications of AI

Game playing
There is some AI in them, but they play well against people mainly through brute force computation--looking at hundreds of thousands of positions. To beat a world champion by brute force and known reliable heuristics requires being able to look at 200 million positions per second.

Speech recognition
In the 1990s, computer speech recognition reached a practical level for limited purposes. Thus United Airlines has replaced its keyboard tree for flight information by a system using speech recognition of flight numbers and city names. It is quite convenient.

Understanding natural language
Just getting a sequence of words into a computer is not enough. Parsing sentences is not enough either. The computer has to be provided with an understanding of the domain the text is about, and this is presently possible only for very limited domains.

Computer vision
The world is composed of three-dimensional objects, but the inputs to the human eye and computers' TV cameras are two dimensional. Some useful programs can work solely in two dimensions, but full computer vision requires partial three-dimensional information that is not just a set of two-dimensional views. At present there are only limited ways of representing three-dimensional information directly, and they are not as good as what humans evidently use.

Expert systems
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. 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.

Intelligent Robots − Robots are able to perform the tasks given by a human. They have sensors to detect physical data from the real world such as light, heat, temperature, movement, sound, bump, and pressure. They have efficient processors, multiple sensors and huge memory, to exhibit intelligence. In addition, they are capable of learning from their mistakes and they can adapt to the new environment.

Heuristic classification

One of the most feasible kinds of expert system given the present knowledge of AI is to put some information in one of a fixed set of categories using several sources of information. An example is advising whether to accept a proposed credit card purchase. Information is available about the owner of the credit card, his record of payment and also about the item he is buying and about the establishment from which he is buying it (e.g., about whether there have been previous credit card frauds at this establishment).

1.3 History of AI

Here is the history of AI during 20th century −

Year	Milestone / Innovation
1943	Foundations for neural networks laid.
1945	Isaac Asimov, a Columbia University alumni, coined the term *Robotics*.
1950	Alan Turing introduced Turing Test for evaluation of intelligence and published *Computing Machinery and Intelligence.* Claude Shannon published *Detailed Analysis of Chess Playing* as a search.
1956	John McCarthy coined the term *Artificial Intelligence*. Demonstration of the first running AI program at Carnegie Mellon University.
1958	John McCarthy invents LISP programming language for AI.
1969	Scientists at Stanford Research Institute Developed *Shakey*, a robot, equipped with locomotion, perception, and problem solving.
1973	The Assembly Robotics group at Edinburgh University built *Freddy*, the Famous Scottish Robot, capable of using vision to locate and assemble models.
1979	The first computer-controlled autonomous vehicle, Stanford Cart, was built.
1990	Major advances in all areas of AI – • Significant demonstrations in machine learning • Case-based reasoning • Multi-agent planning • Scheduling • Data mining, Web Crawler

	• natural language understanding and translation • Vision, Virtual Reality • Games
1997	The Deep Blue Chess Program beats the then world chess champion, Garry Kasparov.
2000	Interactive robot pets become commercially available. MIT displays *Kismet*, a robot with a face that expresses emotions.

2. Problem Solving

There are four things that are to be followed in order to solve a problem:

1. *Define the problem*. A precise definition that must include precise specifications of what the initial situation(s) will be as well as what final situations constitute acceptable solutions to the problem.
2. *Problem must be analyzed*. Some of the important features land up having an immense impact on the appropriateness of various possible techniques for solving the problem.
3. *Isolate and represent the task knowledge* that is necessary to solve the problem.
4. Amongst the available ones *choose the best problem-solving technique*(s) and apply the same to the particular problem.

2.1 Defining the Problem as state Search

To understand what exactly artificial intelligence is, we illustrate some common problems. Problems dealt with in artificial intelligence generally use a common term called 'state'. A state represents a status of the solution at a given step of the problem solving procedure. The solution of a problem, thus, is a collection of the problem states.

The problem solving procedure applies an operator to a state to get the next state. Then it applies another operator to the resulting state to derive a new state. The process of applying an operator to a state and its subsequent transition to the next state, thus, is continued until the goal (desired) state is derived. Such a method of solving a problem is generally referred to as state space approach For example, in order to solve the problem play a game, which is restricted to two person table or board games, we require the rules of the game and the targets for winning as well as a means of representing positions in the game.

The opening position can be defined as the initial state and a winning position as a goal state, there can be more than one. legal moves allow for transfer from initial state to other states leading to the goal state. However the rules are far too copious in most games especially chess where they exceed the number of particles in the universe 10. Thus the rules cannot in general be supplied accurately and computer programs cannot easily handle them. The storage also presents another problem but searching can be achieved by hashing. The number of rules that are used must be minimized and the set can be produced by expressing each rule in as general a form as possible.

The representation of games in this way leads to a state space representation and it is natural for

well organized games with some structure. This representation allows for the formal definition of a problem which necessitates the movement from a set of initial positions to one of a set of target positions. It means that the solution involves using known techniques and a systematic search. This is quite a common method in AI.

Example:

A Water Jug Problem: *We give you two jugs with a maximum capacity of 4-gallon and 3-gallon each and a pump to fill each of the jugs. Neither have any measuring markers on it. Now your job is to get exactly 2 gallons of water into the 4-gallon jug. How will you do that?*

We start with describing the state space as the set of ordered pairs of integers *(x, y), viz. x* = 0, 1, 2, 3, or 4 and *y* = 0, 1,2, or 3; where *x and y* represents the quantity of water in the 4-gallon jug and 3-gallon jug respectively.

The start state is (0, 0) indicating that both the jugs are empty initially. (2, *n)* is our required goal state. Here we do not consider any constant value for y since the problem does not specify the quantity in the 3-gallon jug.

The descriptions of the operators used to solve the problem are shown below. As in the chess problem, they are represented as rules whose left sides are matched against the current state and whose right sides describe the new state that results from applying the rule.

Assumption here were that we can fill a jug from the pump, that we can pour water out of a jug onto the ground, that we can pour water from one jug to another, and that there are no other measuring devices available.

In addition to the problem description given above, all we need, to solve the water jug problem, is a control structure that loops through a simple cycle in which some rule whose left side matches the current state is chosen, the appropriate change to the state is made as described in the corresponding right side, and the resulting state is checked to see if it corresponds to a goal state.

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

Figure : One Solution to Water Jug Problem

1	(x, y) if $x < 4$	$\rightarrow (4, y)$	Fill the 4-gallon jug
2	(x, y) if $y < 3$	$\rightarrow (x, 3)$	Fill the 3-gallon jug
3	(x, y) if $x > 0$	$\rightarrow (x - d, y)$	Pour some water out of the 4-gallon jug
4	(x, y) if $y > 0$	$\rightarrow (x, y - d)$	Pour some water out of the 3-gallon jug
5	(x, y) if $x > 0$	$\rightarrow (0, y)$	Empty the 4-gallon jug on the ground
6	(x, y) if $y > 0$	$\rightarrow (x, 0)$	Empty the 3-gallon jug on the ground
7	(x, y) if $x + y \geq 4$ and $y > 0$	$\rightarrow (4, y - (4 - x))$	Pour water from the 3-gallon jug into the 4-gallon jug until the 4-gallon jug is full
8	(x, y) if $x + y \geq 3$ and $x > 0$	$\rightarrow (x - (3 - y), 3)$	Pour water from the 4-gallon jug into the 3-gallon jug until the 3-gallon jug is full
9	(x, y) if $x + y \leq 4$ and $y > 0$	$\rightarrow (x + y, 0)$	Pour all the water from the 3-gallon jug into the 4-gallon jug
10	(x, y) if $x + y \leq 3$ and $x > 0$	$\rightarrow (0, x + y)$	Pour all the water from the 4-gallon jug into the 3-gallon jug
11	$(0, 2)$	$\rightarrow (2, 0)$	Pour the 2 gallons from the 3-gallon jug into the 4-gallon jug
12	$(2, y)$	$\rightarrow (0, y)$	Empty the 2 gallons in the 4-gallon jug on the ground

Figure : Production Rules for Water Jug Problem

2.2.Problem Formulation

Summarizing what we have just said, in order to provide a formal description of a problem, we must do the following:

1. **Define a state space** that contains all the possible configurations of the relevant objects and perhaps some impossible ones too. It is, of course, possible to define this space without explicitly enumerating all of the states it contains.
2. **Specify initial states**. One or more states within that space that describe possible situations from which the problem-solving process may start.
3. **Specify Goal states**. One or more states that would be acceptable as solutions to the problem.
4. **Specify a set of rules** that describe the actions (operators) available.

Problems which requires search can be formulated by specifying four pieces of information:

- Initial state: *what state are the environment/agent in to begin with?*

- Actions: *the successor function specifies what actions are possible in each state, and what their result would be. It consists of a set of action-state pairs.*
- Goal test: *either an implicit or explicit statement of when the agent's goal has been achieved.*
- Path cost: *a step cost c(x,a,y) for each action 'a' that takes the agent from state 'x' to state 'y' – the sum of all step costs for a sequence of actions is the path cost.*

For example, consider the map of Romania in Figure 2. Let's say that an agent is in the town of *Arad*, and has to goal of getting to *Bucharest*.
What sequence of actions will lead to the agent achieving its goal?

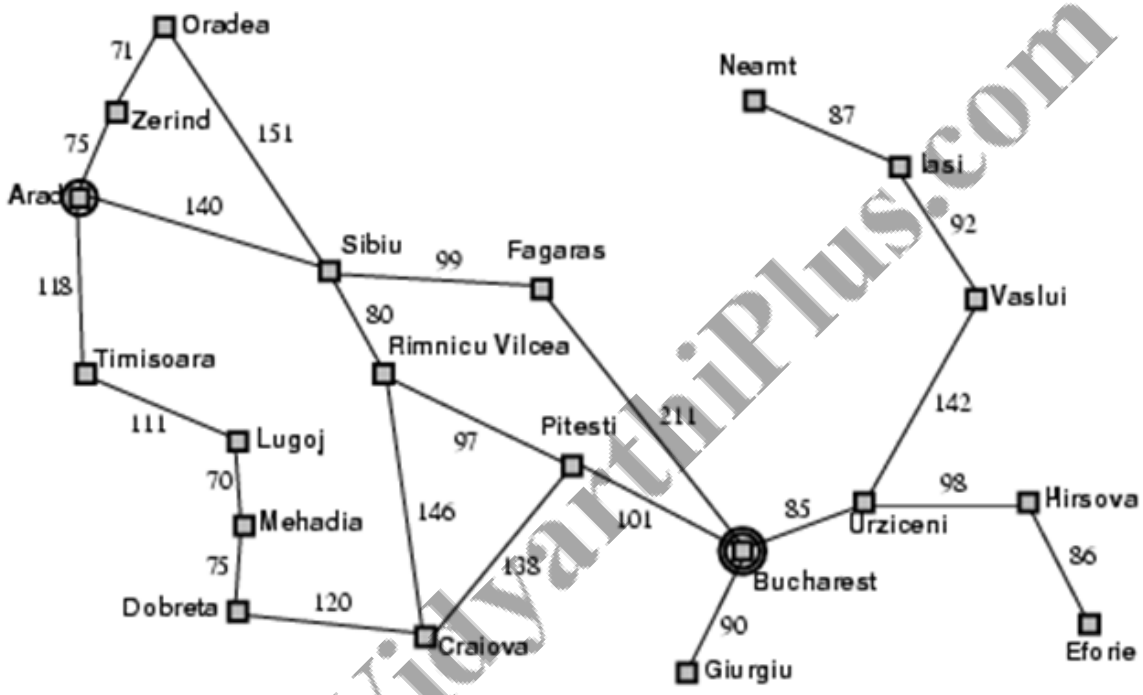


Figure – The state space of the Romania problem

If we assume that the environment is fully-observable and deterministic, then we can formulate this problem as a single-state problem. The environment is fully-observable if the agent knows the map of Romania and his/her current location. It is deterministic if the agent is guaranteed to arrive at the city at the other end of each road it takes. These are both reasonable assumptions in this case. The single-state problem formulation is therefore:

- Initial state: *at Arad*
- Actions: *the successor function S:*
  - *S(Arad) = {<Arad→Zerind, Zerind>, <Arad→Sibiu, Sibiu>, <Arad→Timisoara, Timisoara}*
  - *S(Sibiu) = {<Sibiu→Arad, Arad>, <Sibiu→Oradea, Oradea>, <Sibiu→Fagaras, Fagaras>, Sibiu→Rimnicu Vilcea, Rimnicu Vilcea>}*
  - *etc.*
- Goal test: *at Bucharest*
- Path cost:
  - *c(Arad, Arad→Zerind, Zerind) = 75, c(Arad, Arad→Sibiu, Sibiu) = 140, c(Arad, Arad→Timisoara, Timisoara) = 118, etc.*

Notice the form of the successor function *S*. This specifies an *action-state* pair for each allowable action in each possible state. For example, if the agent is in the state *Arad*, there are 3 possible actions, *Arad→Zerind*, *Arad→Sibiu* and *Arad→Timisoara*, resulting in the states *Zerind*, *Sibiu* and *Timisoara* respectively.

2.3 Production Systems

Production systems provide search structures that forms the core of many intelligent processes. Hence it is useful to structure AI programs in a way that facilitates describing and performing the search process. Do not be confused by other uses of the word *production*, such as to describe what is done in factories. A *production system* consists of:

1. **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.
2. **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.
3. **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.
4. **A rule applier**.

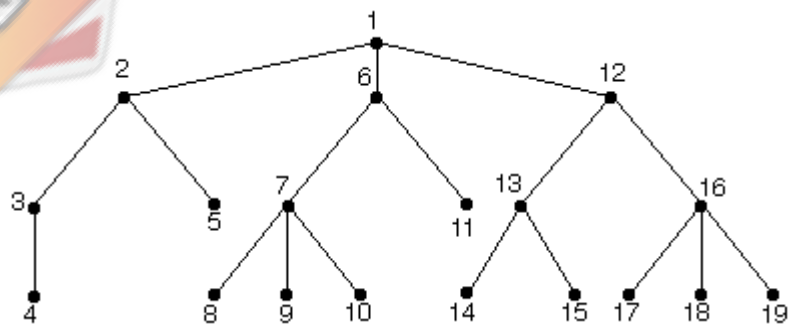
2.4 Control Strategies – Search Strategies

There are 2 basic ways to perform a search:

- Blind search -- can only move according to position in search.
- Heuristic search -- use *domain-specific* information to decide where to search next.

Blind Search Depth-First Search

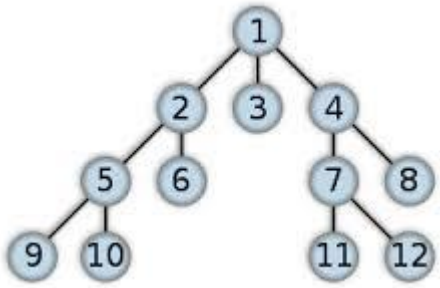
1. Set *L* to be a list of the initial nodes in the problem.
2. If *L* is empty, *fail* otherwise pick the first node *n* from *L*
3. If *n* is a goal state, quit and return path from initial node.
4. Otherwise remove *n* from *L* and add to the front of *L* all of *n*'s children. Label each child with its path from initial node. Return to 2.



Note: All numbers in Fig refer to order visited in search.

Breadth-First Search

1. Set *L* to be a list of the initial nodes in the problem.
2. If *L* is empty, *fail* otherwise pick the first node *n* from *L*
3. If *n* is a goal state, quit and return path from initial node.
4. Otherwise remove *n* from *L* and add to the end of *L* all of *n*'s children. Label each child with its path from initial node. Return to 2.



Note: All numbers in Fig refer to order visited in search.

Heuristic Search

A *heuristic* is a method that

- might not always find the best solution
- ***but*** is guaranteed to find a good solution in reasonable time.
- By sacrificing completeness it increases efficiency.
- Useful in solving tough problems which
  - could not be solved any other way.
  - solutions take an infinite time or very long time to compute.

The *classic* example of heuristic search methods is the travelling salesman problem.

3. Problem Characteristics

A problem may have different aspects of representation and explanation. In order to choose the most appropriate method for a particular problem, it is necessary to analyze the problem along several key dimensions.

Heuristic search is a very general method applicable to a large class of problem . It includes a variety of techniques. In order to choose an appropriate method, it is necessary to analyze the problem with respect to the following considerations.

1. Is the problem decomposable ?

A very large and composite problem can be easily solved if it can be broken into smaller problems and recursion could be used. Suppose we want to solve.

Ex:- $\int x2 + 3x+\sin 2x \cos 2x \, dx$

This can be done by breaking it into three smaller problems and solving each by applying specific rules. Adding the results the complete solution is obtained.

2. Can solution steps be ignored or undone?

Problem fall under three classes ignorable , recoverable and irrecoverable. This classification is with reference to the steps of the solution to a problem. Consider thermo proving. We may later find that it is of no help. We can still proceed further, since nothing is lost by this redundant step. This is an example of ignorable solutions steps.

Now consider the 8 puzzle problem tray and arranged in specified order. While moving from the start state towards goal state, we may make some stupid move and consider theorem proving. We may proceed by first proving lemma. But we may backtrack and undo the unwanted move. This only involves additional steps and the solution steps are recoverable.

Lastly consider the game of chess. If a wrong move is made, it can neither be ignored nor be recovered. The thing to do is to make the best use of current situation and proceed. This is an example of an irrecoverable solution steps.

1. Ignorable problems Ex:- theorem proving
   · In which solution steps can be ignored.
2. Recoverable problems Ex:- 8 puzzle
   · In which solution steps can be undone
3. Irrecoverable problems Ex:- Chess
   · In which solution steps can't be undone

A knowledge of these will help in determining the control structure.

3.. Is the Universal Predictable?

Problems can be classified into those with certain outcome (eight puzzle and water jug problems) and those with uncertain outcome (playing cards) . in certain – outcome problems, planning could be done to generate a sequence of operators that guarantees to a lead to a solution. Planning helps to avoid unwanted solution steps. For uncertain out come problems, planning can at best generate a sequence of operators that has a good probability of leading to a solution. The uncertain outcome problems do not guarantee a solution and it is often very expensive since the number of solution and it is often very expensive since the number of solution paths to be explored increases exponentially with the number of points at which the outcome can not be predicted. Thus one of the hardest types of problems to solve is the irrecoverable, uncertain – outcome problems (Ex:- Playing cards).

4. Is good solution absolute or relative ?

(Is the solution a state or a path ?)

There are two categories of problems. In one, like the water jug and 8 puzzle problems, we are satisfied with the solution, unmindful of the solution path taken, whereas in the other category not just any solution is acceptable. We want the best, like that of traveling sales man problem, where it is the shortest path. In any – path problems, by heuristic methods we obtain a solution and we do not explore alternatives. For the best-path problems all possible paths are explored using an exhaustive search until the best path is obtained.

5. The knowledge base consistent ?

In some problems the knowledge base is consistent and in some it is not. For example consider the case when a Boolean expression is evaluated. The knowledge base now contains theorems and laws of Boolean Algebra which are always true. On the contrary consider a knowledge base that contains facts about production and cost. These keep varying with time. Hence many reasoning schemes that work well in consistent domains are not appropriate in inconsistent domains.

Ex.Boolean expression evaluation.

6. What is the role of Knowledge?

Though one could have unlimited computing power, the size of the knowledge base available for solving the problem does matter in arriving at a good solution. Take for example the game of playing chess, just the rues for determining legal moves and some simple control mechanism is sufficient to arrive at a solution. But additional knowledge about good strategy and tactics could help to constrain the search and speed up the execution of the program. The solution would then be realistic.

Consider the case of predicting the political trend. This would require an enormous amount of knowledge even to be able to recognize a solution , leave alone the best.

Ex:- 1. Playing chess 2. News paper understanding

7. Does the task requires interaction with the person.

The problems can again be categorized under two heads.

1. Solitary in which the computer will be given a problem description and will produce an answer, with no intermediate communication and with he demand for an explanation of the reasoning process. Simple theorem proving falls under this category . given the basic rules and laws, the theorem could be proved, if one exists.

Ex:- theorem proving (give basic rules & laws to computer)

2. Conversational, in which there will be intermediate communication between a person and the computer, wither to provide additional assistance to the computer or to provide additional informed information to the user, or both problems such as medical diagnosis fall under this category, where people will be unwilling to accept the verdict of the program, if they can not follow its reasoning.

Ex:- Problems such as medical diagnosis.

The above characteristics of a problem are called as 7-problem characteristics under which the solution must take place.

Example : Problem Characteristics for Chess game

Problem characteristic	Satisfied	Reason
Is the problem decomposable?	No	One game have Single solution
Can solution steps be ignored or undone?	No	In actual game(not in PC) we can't undo previous steps
Is the problem universe predictable?	No	Problem Universe is not predictable as we are not sure about move of other player(second player)
Is a good solution absolute or relative?	absolute	Absolute solution : once you get one solution you do need to bother about other possible solution. Relative Solution : once you get one solution you have to find another possible solution to check which solution is best(i.e low cost). By considering this **chess is absolute**
Is the solution a state or a path?	Path	Is the solution a state or a path to a state? – For natural language understanding, some of the words have different interpretations .therefore sentence may cause ambiguity. To solve the problem we need to find interpretation only , the workings are not necessary (i.e path to solution is not necessary) So In chess winning state(goal state) describe path to state
What is the role of knowledge?		lot of knowledge helps to constrain the search for a solution.
Does the task require human-interaction?	No	Conversational In which there is intermediate communication between a person and the computer, either to provide additional assistance to the computer or ***to provide additional information to the user***, or both. In chess additional assistance is not required

3.1 Problem Classification

Actual problems are examined from the point of view , the task here is examine an input and decide which of a set of known classes.

Ex:- Problems such as medical diagnosis , engineering design.

4. Production System

The production system is a model of computation that can be applied to implement search algorithms and model human problem solving. Such problem solving knowledge can be packed up in the form of little quanta called productions. A production is a rule consisting of a situation

recognition part and an action part. A production is a situation-action pair in which the left side is a list of things to watch for and the right side is a list of things to do so. When productions are used in deductive systems, the situation that trigger productions are specified combination of facts. The actions are restricted to being assertion of new facts deduced directly from the triggering combination. Production systems may be called premise conclusion pairs rather than situation action pair.

A production system consists of following components.

(a) *A set of production rules*, which are of the form A□B. Each rule consists of left hand side constituent that represent the current problem state and a right hand side that represent an output state. A rule is applicable if its left hand side matches with the current problem state.

(b) *A database*, which contains all the appropriate information for the particular task. Some part of the database may be permanent while some part of this may pertain only to the solution of the current problem.

(c) *A control strategy* that specifies order in which the rules will be compared to the database of rules and a way of resolving the conflicts that arise when several rules match simultaneously.

(d) *A rule applier*, which checks the capability of rule by matching the content state with the left hand

side of the rule and finds the appropriate rule from database of rules.

The important roles played by production systems include a powerful knowledge representation scheme. A production system not only represents knowledge but also action. It acts as a bridge between AI and expert systems. Production system provides a language in which the representation of expert knowledge is very natural. We can represent knowledge in a production system as a set of rules of the form

If (condition) THEN (condition)

along with a control system and a database. The control system serves as a rule interpreter and sequencer. The database acts as a context buffer, which records the conditions evaluated by the rules and information on which the rules act. The production rules are also known as condition – action, antecedent – consequent, pattern – action, situation – response, feedback – result pairs.

For example,

If (you have an exam tomorrow)

THEN (study the whole night)

The production system can be classified as monotonic, non-monotonic, partially commutative and commutative.

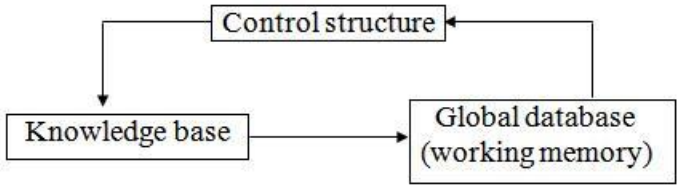


Figure: Architecture of Production System

4.1 Production System Characteristics

Production systems are important in building intelligent matches which can provide us a good set of production rules, for solving problems.

There are four types of production system characteristics, namely

1. Monotonic production system
2. Non-monotonic production system
3. Commutative law based production system, and lastly,
4. Partially commutative law based production system.

Monotonic Production System (MPS): The Monotonic production system (MPS) is a system in which the application of a rule never prevents later application of the another rule that could also have been applied at the time that the first rule was selected.

Non-monotonic Production (NMPS): The non-monotonic production system is a system in which the application of a rule prevents the later application of the another rule which may not have been applied at the time that the first rule was selected, i.e. it is a system in which the above rule is not true, i.e. the monotonic production system rule not true.

Commutative Law Based Production System (CBPS): Commutative law based production systems is a system in which it satisfies both monotonic & partially commutative.

Partially Commutative Production System (PCPS): The partially commutative production system is a system with the property that if the application of those rules that is allowable & also transforms from state x to state ‘y’.

5. Heuristic Search methods

A Heuristic technique helps in solving problems, even though there is no guarantee that it will never lead in the wrong direction. There are heuristics of every general applicability as well as domain specific. The strategies are general purpose heuristics. In order to use them in a specific domain they are coupler with some domain specific heuristics. There are two major ways in which domain - specific, heuristic information can be incorporated into rule-based search procedure.

- In the rules themselves
- As a heuristic function that evaluates individual problem states and determines how desired they are.

A heuristic function is a function that maps from problem state description to measures desirability, usually represented as number weights. The value of a heuristic function at a given node in the search process gives a good estimate of that node being on the desired path to solution. Well designed heuristic functions can provides a fairly good estimate of whether a path is good or not. (" The sum of the distances traveled so far" is a simple heuristic function in the traveling salesman problem) . the purpose of a heuristic function is to guide the search process in the most profitable directions, by suggesting which path to follow first when more than one path is available. However in many problems, the cost of computing the value of a heuristic function would be more than the effort saved in the search process. Hence generally there is a trade-off between the cost of evaluating a heuristic function and the savings in search that the function provides.

5.1 Generate and Test Algorithm

1. generate a possible solution which can either be a point in the problem space or a path from the initial state.
2. test to see if this possible solution is a real solution by comparing the state reached with the set of goal states.
3. if it is a real solution, return. Otherwise repeat from 1.

This method is basically a depth first search as complete solutions must be created before testing. It is often called the British Museum method as it is like looking for an exhibit at random. A heuristic is needed to sharpen up the search. Consider the problem of four 6-sided cubes, and each side of the cube is painted in one of four colours. The four cubes are placed next to one another and the problem lies in arranging them so that the four available colours are displayed whichever way the 4 cubes are viewed. The problem can only be solved if there are at least four sides coloured in each colour and the number of options tested can be reduced using heuristics if the most popular colour is hidden by the adjacent cube.

5.2 Hill climbing

This is a variety of depth-first (generate - and - test) search. A feedback is used here to decide on the direction of motion in the search space. In the depth-first search, the test function will merely accept or reject a solution. But in hill climbing the test function is provided with a heuristic

function which provides an estimate of how close a given state is to goal state. The hill climbing test procedure is as follows :

1. General he first proposed solution as done in depth-first procedure. See if it is a solution. If so quit , else continue.
2. From this solution generate new set of solutions use , some application rules
3. For each element of this set

    (i) Apply test function. It is a solution quit.
    (ii) Else see whether it is closer to the goal state than the solution already generated. If yes, remember it else discard it.

4. Take the best element so far generated and use it as the next proposed solution.
This step corresponds to move through the problem space in the direction
Towards the goal state.
5. Go back to step 2.

Sometimes this procedure may lead to a position, which is not a solution, but from which there is no move that improves things. This will happen if we have reached one of the following three states.

(a) A "**local maximum** " which is a state better than all its neighbors , but is not better than some other states farther away. Local maxim sometimes occur with in sight of a solution. In such cases they are called " Foothills".

(b) A "**plateau**" which is a flat area of the search space, in which neighboring states have the same value. On a plateau, it is not possible to determine the best direction in which to move by making local comparisons.

(c) A "**ridge**" which is an area in the search that is higher than the surrounding areas, but can not be searched in a simple move.

To overcome theses problems we can

(a) Back track to some earlier nodes and try a different direction. This is a good way of dealing with local maxim.

(b) Make a big jump an some direction to a new area in the search. This can be done by applying two more rules of the same rule several times, before testing. This is a good strategy is dealing with plate and ridges. Hill climbing becomes inefficient in large problem spaces, and when combinatorial explosion occurs. But it is a useful when combined with other methods.

Steepest Ascent Hill Climbing

This differs from the basic Hill climbing algorithm by choosing the best successor rather than the first successor that is better. This indicates that it has elements of the breadth first algorithm.

Steepest ascent Hill climbing algorithm
1 Evaluate the initial state
2 If it is goal state Then quit otherwise make the current state this initial state and proceed;
3 Repeat set Target to be the state that any successor of the current state can better; for each operator that can be applied to the current state apply the new operator and create a new state evaluate this state.

- If this state is goal state Then quit. Otherwise compare with Target
- If better set Target to this value
- If Target is better than current state set current state to Target
- Until a solution is found or current state does not change.

Both the basic and this method of hill climbing may fail to find a solution by reaching a state from which no subsequent improvement can be made and this state is not the solution.

Local maximum state is a state which is better than its neighbors but is not better than states faraway. These are often known as foothills. Plateau states are states which have approximately the same value and it is not clear in which direction to move in order to reach the solution. Ridge states are special types of local maximum states.

The surrounding area is basically unfriendly and makes it difficult to escape from, in single steps, and so the path peters out when surrounded by ridges. Escape relies on: backtracking to a previous good state and proceed in a completely different direction--- involves keeping records of the current path from the outset; making a gigantic leap forward to a different part of the search space perhaps by applying a sensible small step repeatedly, good for plateaux; applying more than one rule at a time before testing, good for ridges.
None of these escape strategies can guarantee success.

5.3 Simulated Annealing

This is a variation on hill climbing and the idea is to include a general survey of the scene to avoid climbing false foot hills.

The whole space is explored initially and this avoids the danger of being caught on a plateau or ridge and makes the procedure less sensitive to the starting point. There are two additional changes; we go for minimisation rather than creating maxima and we use the term objective function rather than heuristic. It becomes clear that we are valley descending rather than hill climbing. The title comes from the metallurgical process of heating metals and then letting them cool until they reach a minimal energy steady final state. The probability that the metal will jump to a higher energy level is given by $p = \exp^{-\Delta E/kT}$ where k is Boltzmann's constant. The rate at which the system is cooled is called the annealing schedule. $\Delta E$is called the change in the value of the objective function and kT is called T a type of temperature. An example of a problem suitable for such an algorithm is the travelling salesman. The SIMULATED ANNEALING algorithm is based upon the physical process which occurs in metallurgy where metals are heated to high temperatures and are then cooled. The rate of cooling clearly affects the finished product. If the rate of cooling is fast, such as when the metal is quenched in a large tank

of water the structure at high temperatures persists at low temperature and large crystal structures exist, which in this case is equivalent to a local maximum. On the other hand if the rate of cooling is slow as in an air based method then a more uniform crystalline structure exists equivalent to a global maximum.The probability of making a large uphill move is lower than a small move and the probability of making large moves decreases with temperature. Downward moves are allowed at any time.

1. Evaluate the initial state.
2. If it is goal state Then quit otherwise make the current state this initial state and proceed.
3. Make variable *BEST_STATE* to current state
4. Set temperature, *T*, according to the annealing schedule
5. Repeat
   ΔE-- difference between the values of current and new states
   1. If this new state is goal state Then quit
   2. Otherwise compare with the current state
   3. If better set *BEST_STATE* to the value of this state and make the current the new state
   4. If it is not better then make it the current state with probability *p*'. This involves generating a random number in the range 0 to 1 and comparing it with a half, if it is less than a half do nothing and if it is greater than a half accept this state as the next current be a half.
   5. Revise T in the annealing schedule dependent on number of nodes in tree

   Until a solution is found or no more new operators
6. Return *BEST_STATE* as the answer

5.4 Best First Search

A combination of depth first and breadth first searches.

Depth first is good because a solution can be found without computing all nodes and breadth first is good because it does not get trapped in dead ends. The best first search allows us to switch between paths thus gaining the benefit of both approaches. At each step the most promising node is chosen. If one of the nodes chosen generates nodes that are less promising it is possible to choose another at the same level and in effect the search changes from depth to breadth. If on analysis these are no better then this previously unexpanded node and branch is not forgotten and the search method reverts to the descendants of the first choice and proceeds, backtracking as it were.

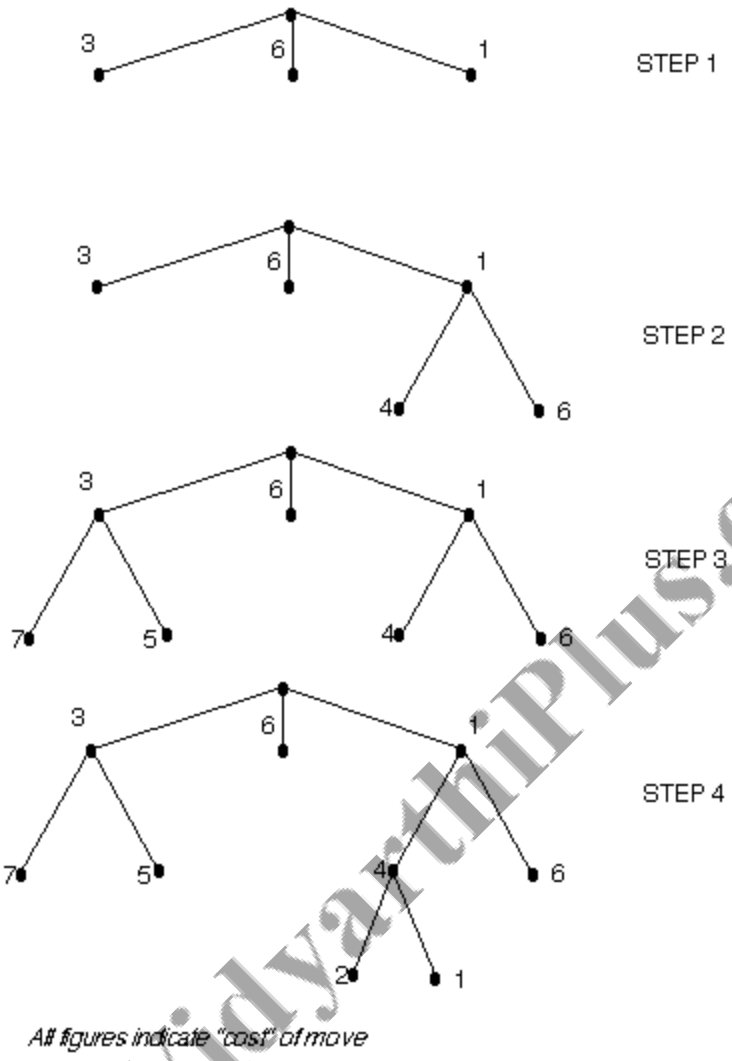
This process is very similar to steepest ascent, but in hill climbing once a move is chosen and the others rejected the others are never reconsidered whilst in best first they are saved to enable revisits if an impasse occurs on the apparent best path. Also the best available state is selected in best first even its value is worse than the value of the node just explored whereas in hill climbing the progress stops if there are no better successor nodes. The best first search algorithm will involve an OR graph which avoids the problem of node duplication and assumes that each node has a parent link to give the best node from which it came and a link to all its successors. In this way if a better node is found this path can be propagated down to the successors. This method of using an OR graph requires 2 lists of nodes .

OPEN is a priority queue of nodes that have been evaluated by the heuristic function but which have not yet been expanded into successors. The most promising nodes are at the front. CLOSED are nodes that have already been generated and these nodes must be stored because a graph is being used in preference to a tree.

Heuristics In order to find the most promising nodes a heuristic function is needed called *f'* where *f'* is an approximation to *f* and is made up of two parts *g* and *h'* where *g* is the cost of going from the initial state to the current node; *g* is considered simply in this context to be the number of arcs traversed each of which is treated as being of unit weight. *h'* is an estimate of the initial cost of getting from the current node to the goal state. The function *f'* is the approximate value or estimate of getting from the initial state to the goal state. Both *g* and *h'* are positive valued variables. Best First The Best First algorithm is a simplified form of the *A** algorithm. From *A** we note that *f' = g+h'* where *g* is a measure of the time taken to go from the initial node to the current node and *h'* is an estimate of the time taken to solution from the current node. Thus *f'* is an estimate of how long it takes to go from the initial node to the solution. As an aid we take the time to go from one node to the next to be a constant at 1.

Best First Search Algorithm:

1. Start with OPEN holding the initial state
2. Pick the best node on OPEN
3. Generate its successors
4. For each successor Do
    - If it has not been generated before evaluate it add it to OPEN and record its parent
    - If it has been generated before change the parent if this new path is better and in that case update the cost of getting to any successor nodes
5. If a goal is found or no more nodes left in OPEN, quit, else return to 2.



5.5 The *A Algorithm**

A heuristic function f estimates the merits of each generated node. This function f has two components g and h. the function g gives the cost of getting from the initial state to the current node. The function h is an estimate of the addition cost of getting from current node to a goal state. The function f (=g+h) gives the cost of getting from the initial state to a goal state via the current node.

THE A* ALGORITHM:-

1. Start with OPEN containing the initial node. Its g=0 and f ' = h '

    Set CLOSED to empty list.

2. Repeat

    If OPEN is empty , stop and return failure

Else pick the BESTNODE on OPEN with lowest f ' value and place it on CLOSED

If BESTNODE is goal state return success and stop

Else

Generate the successors of BESTNODE.

For each SUCCESSOR do the following:

1. Set SUCCESSOR to point back to BESTNODE. (back links will help to recover the path)

2. compute g(SUCCESSOR) = g(BESTNODE) cost of getting from BESTNODE to SUCCESSOR.

3. If SUCCESSOR is the same as any node on OPEN, call that node OLS and add OLD to BESTNODE 's successors. Check g(OLD) and g(SUCCESSOR). It g(SUCCESSOR) is cheaper then reset OLD 's parent link to point to BESTNODE. Update g(OLD) and f '(OLD).

4. If SUCCESSOR was not on OPEN , see if it is on CLOSED . if so call the node CLOSED OLD , and better as earlier and set the parent link and g and f ' values appropriately.

5. If SUCCESSOR was not already on earlier OPEN or CLOSED, then put it on OPEN and add it to the list of BESTNODE 's successors.

Compute f ' (SUCCESSOR) = g(SUCCESSOR) + h ' (SUCCESSOR)

Best first searches will always find good paths to a goal after exploring the entire state space. All that is required is that a good measure of goal distance be used.

Graceful decay of admissibility

If *h*' rarely overestimates h by more than d then the *A** algorithm will rarely find a solution whose cost is *d* greater than the optimal solution.

5.6 PROBLEM REDUCTION (AND - OR graphs - AO * Algorithm)

When a problem can be divided into a set of sub problems, where each sub problem can be solved separately and a combination of these will be a solution, AND-OR graphs or AND - OR trees are used for representing the solution. The decomposition of the problem or problem reduction generates AND arcs. One AND are may point to any number of successor nodes. All these must be solved so that the arc will rise to many arcs, indicating several possible solutions. Hence the graph is known as AND - OR instead of AND. Figure shows an AND - OR graph.

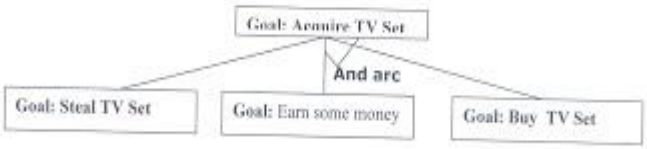


Figure shows AND - Or graph - an example.

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

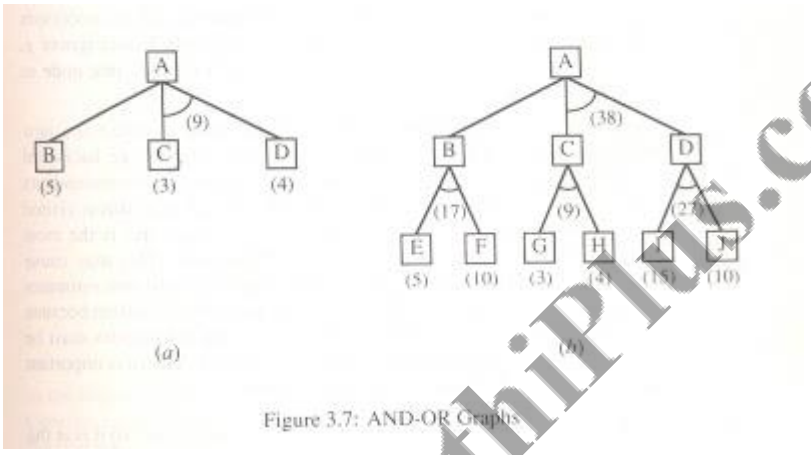


Figure 3.7: AND-OR Graphs

FIGURE : AND - OR graph

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

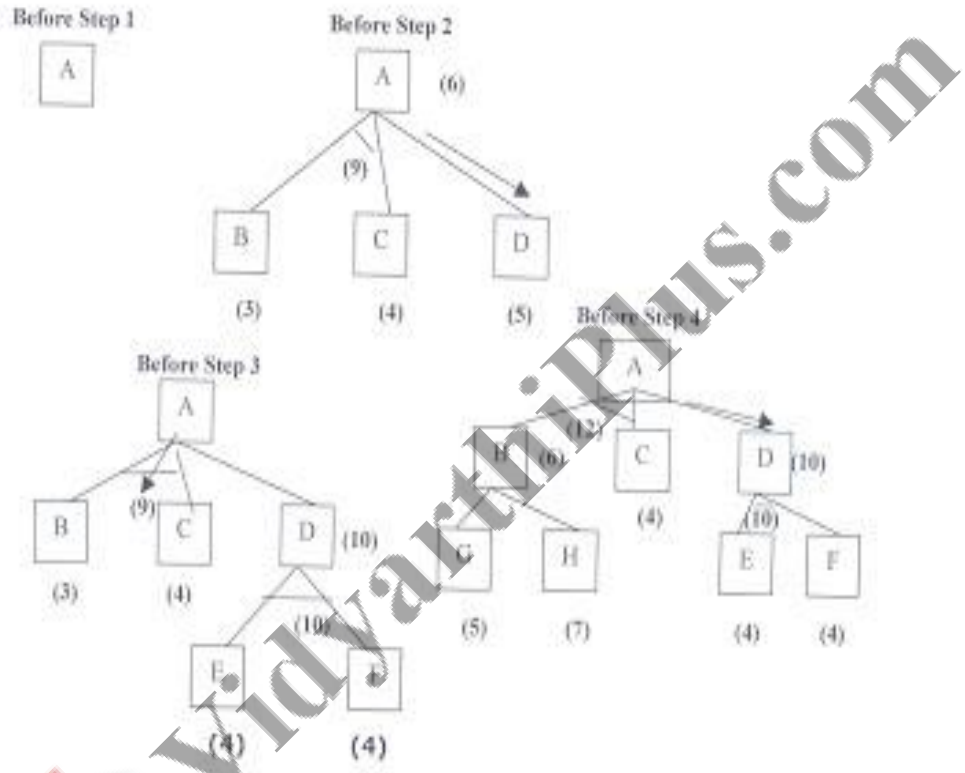
Thus the choice of the next node to expand depends not only n a value but also on whether that node is part of the current best path form the initial mode. Figure (b) makes this clearer. In figure the node G appears to be the most promising node, with the least f ' value. But G is not on the current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

1. traverse the graph starting at the initial node and following the current best path, and accumulate the set of nodes that are on the path and have not yet been expanded.

2. Pick one of these unexpanded nodes and expand it. Add its successors to the graph and computer f ' (cost of the remaining distance) for each of them.

3. Change the f ' estimate of the newly expanded node to reflect the new information produced by its successors. Propagate this change backward through the graph. Decide which of the current best path.

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



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

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

FUTILITY is used. The estimated cost of a solution is greater than FUTILITY then the search is abandoned as too expansive to be practical.
For representing above graphs AO* algorithm is as follows

AO* ALGORITHM:

1. Let G consists only to the node representing the initial state call this node INIT. Compute h' (INIT).

2. Until INIT is labeled SOLVED or hi (INIT) becomes greater than FUTILITY, repeat the following procedure.

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

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

(a) add SUCCESSOR to graph G

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

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

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

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

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

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

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

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

AO* Search Procedure.

1. Place the start node on open.

2. Using the search tree, compute the most promising solution tree TP .

3. Select node n that is both on open and a part of tp, remove n from open and place it no closed.

4. If n is a goal node, label n as solved. If the start node is solved, exit with success where tp is the solution tree, remove all nodes from open with a solved ancestor.

5. If n is not solvable node, label n as unsolvable. If the start node is labeled as unsolvable, exit with failure. Remove all nodes from open ,with unsolvable ancestors.

6. Otherwise, expand node n generating all of its successor compute the cost of for each newly generated node and place all such nodes on open.

7. Go back to step(2)

Note: AO* will always find minimum cost solution.

Means-Ends Analysis

- Allows both backward and forward searching.
- This means we could solve major parts of a problem first and then return to smaller problems when assembling the final solution.
- GPS was the first AI program to exploit means-ends analysis.
- STRIPS (A robot Planner) is an advanced problem solver that incorporates means-ends analysis and other techniques.

Very loosely the means-ends analysis algorithm is:

1. Until the goal is reached or no more procedures are available:
    - Describe the current state, the goal state and the differences between the two.
    - Use the difference the describe a procedure that will hopefully get nearer to goal.
    - Use the procedure and update current state.
2. If goal is reached then **success** otherwise **fail**.

5.7 Constraint Satisfaction

- The general problem is to find a solution that satisfies a set of constraints.
- heuristics used not to estimate *the distance to the goal* but to decide what node to expand nest.
- Examples of this technique are design problem, labelling graphs, robot path planning and cryptarithmetic puzzles (see last year).

Algorithm:

1. Propagate available constraints:

- Open all objects that must be assigned values in a complete solution.
- Repeat until inconsistency or all objects assigned valid values:
  - select an object and strengthen as much as possible the set of constraints that apply to object.
  - If set of constraints different from previous set then open all objects that share any of these constraints.
  - remove selected object.

2. If union of constraints discovered above defines a solution return solution.
3. If union of constraints discovered above defines a contradiction return failure
4. Make a *guess* in order to proceed. Repeat until a solution is found or all possible solutions exhausted:

   - Select an object with a no assigned value and try to strengthen its constraints.
   - recursively invoke constraint satisfaction with the current set of constraints plus the selected strengthening constraint.

   (*** use map coloring problem discussed in the class***)

5.8 Why are these topics important?

- Search and knowledge representation form the basis for many AI techniques.
- Here are a few pointers as to where specific search methods are used.

Knowledge Representation
-- Best first search (A*), Constraint satisfaction and means-ends analysis searches used in Rule based and knowledge.

Uncertain reasoning
-- Depth first, breadth first and constraint satisfaction methods used.

Distributed reasoning
-- Best first search (A*) and Constraint satisfaction.

Planning
-- Best first search (A*), AO*, Constraint satisfaction and means-ends analysis.

Understanding
-- Constraint satisfaction.

Learning
-- Constraint satisfaction, Means-ends analysis.

Common sense
-- Constraint satisfaction.

Vision
-- depth first, breadth first, heuristics, simulated annealing, constraint satisfaction are all used extensively.

Robotics
-- Constraint satisfaction and means-ends analysis used in planning robot paths.

Rajalakshmi Institute of Technology
Department of Information Technology

CS6659 - Artificial Intelligence

UNIT II REPRESENTATION OF KNOWLEDGE

Game playing – Knowledge representation, Knowledge representation using Predicate logic, Introduction to predicate calculus, Resolution, Use of predicate calculus, Knowledge representation using other logic-Structured representation of knowledge.

1. Game Playing

Introduction

Game playing has been a major topic of AI since the very beginning. Beside the attraction of the topic to people, it is also because its close relation to "intelligence", and its well-defined states and rules.

The most common used AI technique in game is search. In some other problem-solving activities, state change is solely caused by the action of the system itself. However, in multi-player games, states also depend on the actions of other players (systems) who usually have different goals.

A special situation that has been studied most is two-person zero-sum game, where the two players have exactly opposite goals, that is, each state can be evaluated by a score from one player's viewpoint, and the other's viewpoint is exactly the opposite. This type of game is common, and easy to analyze, though not all competitions are zero-sum!

There are *perfect information games* (such as Chess and Go) and *imperfect information games* (such as Bridge and games where dice are used). Given sufficient time and space, usually an optimum solution can be obtained for the former by exhaustive search, though not for the latter. However, for most interesting games, such a solution is usually too inefficient to be practically used.

1.1. Minimax Procedure

For two-person zero-sum perfect-information game, if the two players take turn to move, the minimax procedure can solve the problem given sufficient computational resources. This algorithm assumes each player takes the best move in each step.

First, we distinguish two types of nodes, MAX and MIN, in the state graph, determined by the depth of the search tree.

Minimax procedure: starting from the leaves of the tree (with final scores with respect to one player, MAX), and go backwards towards the root (the starting state).

At each step, one player (MAX) takes the action that leads to the highest score, while the other player (MIN) takes the action that leads to the lowest score.

All nodes in the tree will all be scored, and the path from root to the actual result is the one on which all nodes have the same score.

Example:

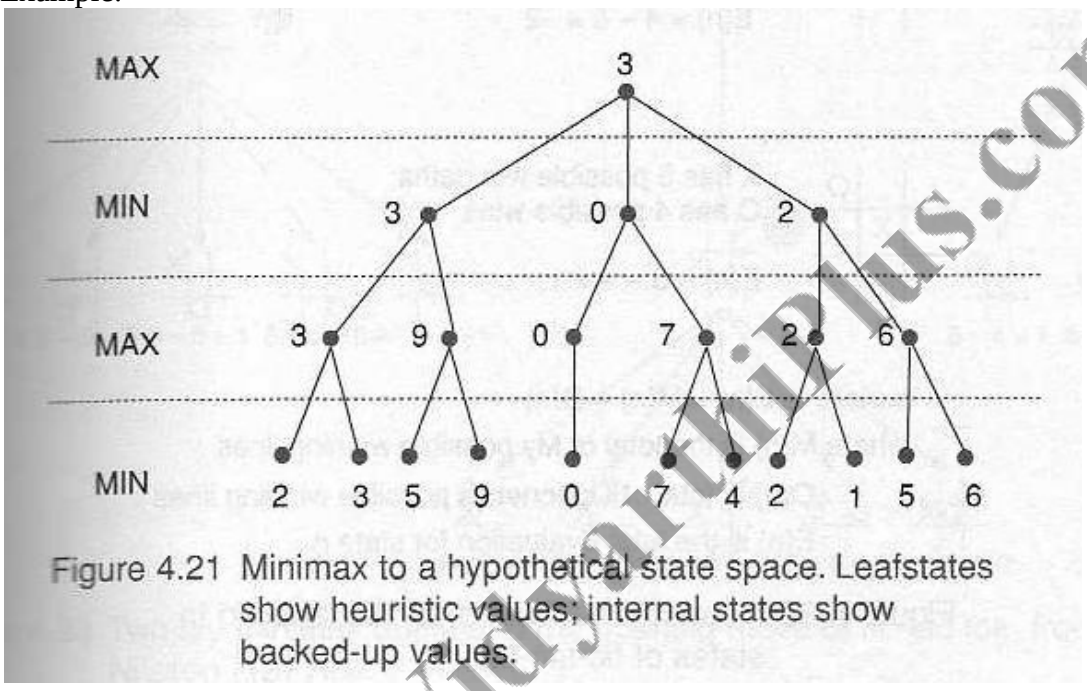


Figure 4.21 Minimax to a hypothetical state space. Leafstates show heuristic values; internal states show backed-up values.

Because of computational resources limitation, the search depth is usually restricted, and estimated scores generated by a heuristic function are used in place of the actual score in the above procedure.

Example: Tic-tac-toe, with the difference of possible win paths as the henristic function.

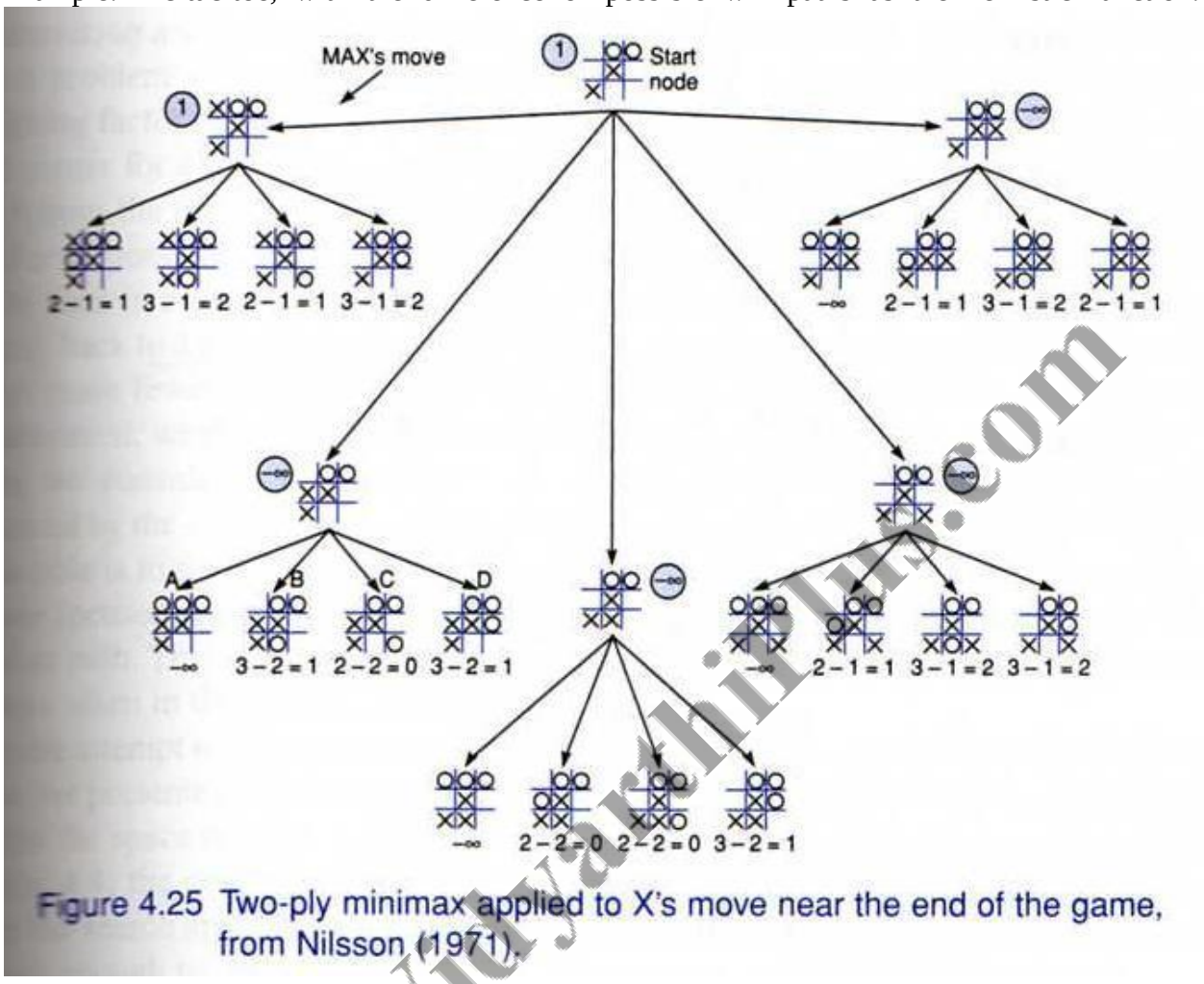


Figure 4.25 Two-ply minimax applied to X's move near the end of the game, from Nilsson (1971).

1.2 Alpha-Beta Pruning

Very often, the game graph does not need to be fully explored using Minimax.

Based on explored nodes' score, inequity can be set up for nodes whose children haven't been exhaustively explored. Under certain conditions, some branches of the tree can be ignored without changing the final score of the root.

In Alpha-Beta Pruning, each MAX node has an *alpha* value, which never decreases; each MIN node has a *beta* value, which never increases. These values are set and updated when the value of a child is obtained. Search is depth-first, and stops at any MIN node whose *beta* value is smaller than or equal to the *alpha* value of its parent, as well as at any MAX node whose *alpha* value is greater than or equal to the *beta* value of its parent.

Examples: in the following partial trees, the other children of node (5) do not need to be generated.

```
(1)MAX[>=3] ----- (2)MIN[==3] ----- (3)MAX[==5]
   |              |------------ (4)MAX[==3]
   |
   |------------ (5)MIN[<=0] ----- (6)MAX[==0]
                 |-----------X
                 |-----------X
```

```
(1)MIN[<=5] ----- (2)MAX[==5] ----- (3)MIN[==5]
   |              |------------ (4)MIN[==3]
   |
   |------------ (5)MAX[>=8] ----- (6)MIN[==8]
                 |-----------X
                 |-----------X
```

2. Knowledge Representation.

solving complex AI problems requires large amounts of knowledge and mechanisms for manipulating that knowledge. The inference mechanisms that operate on knowledge, relay on the ways knowledge is represented. A good knowledge representation model allows for more powerful inference mechanisms that operate on them. While representing knowledge one has to consider two things.

1. **Facts**, which are truths in some relevant world.

2. **Representation of facts** in some chosen formalism. These are the things

Which are actually manipulated by inference mechanism.

Knowledge representation schemes are useful only if there are functions that map facts to representations and vice versa. AI is more concerned with a natural language representation of facts and the functions which map natural language sentences into some representational formalism. An appealing way of representing facts is using the language of logic. Logical formalism provides a way of deriving new knowledge from the old through mathematical deduction. In this formalism, we can conclude that a new statement is true by proving that it follows from the statements already known to be facts.

2.1 Representing Simple Facts in Logic.

Symbolic Logic was being used to represent knowledge even before the advent of digital computers. Today First Order Predicate Logic (FOPL) or simple Predicate Logic plays an important role in AI for the representation of knowledge. A familiarity with FOPL is important. It offers the only formal approach to r4easoning that has a sound theoretical foundation, this is important in the attempts to automate the reasoning process, because inference should be correct and logically sound. The structure of FOPL is flexible and it permits accurate representation of natural language reasonably well.

Logic is a formal method for 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 . in predicate logic statements from a natural language like English are

translated into symbolic structures comprised of predicates, functions, variables, constants, quantifiers and logical connectives. The symbols form the basic building

blocks for the knowledge, and their combination into valid structures is accomplished using the syntax of FOPL. Once structures have been created to represent basic facts, inference rules may then be applied to compare, combine and transform these “assumed” structures into new “deduced” structures. This is how automated reasoning or inferencing is performed.

{ The following standard logic symbols are used in knowledge representation:

→	for " implies " or " then"
~	for "not " or "negation"
^	for "and" or "Conjunction
V	for "or" or "disjunction"
↔	for "iff" or "Double implication"
∀	for " forall"
∃	for "there exists" }

as a simple example of use of logic, the statement
"All employees of the ABC company are programmers" might be written in predicate logic as

$(\forall x)$ (ABC *Co*_Emp(x) → programmer(x)).

∀ x is read as "for all x". the predicates ABC_Co_Emp(x) and programmer(x) are read as "if x is an ABC_Co" and "x is a programmer", respectively . the symbol x is a variable and it can assume a person's name . if it is also known that Ram is employee of ABC company , one can draw the condition that Ram is a programmer.

This example shows how natural language sentences can be translated into predicate logic statements. Once translated, such statements can be put into a knowledge base and subsequently used in a program to perform inferencing.

Structured Representation of Knowledge.

2.2 STRUCTURED REPRESNTATION OF KNOWLEDGE

Representing knowledge using logical formalism, like predicate logic, has several advantages. They can be combined with powerful inference mechanisms like resolution, which makes reasoning with facts easy. But using logical formalism complex structures of the world, objects and their relationships, events, sequences of events etc. can not be described easily.

A good system for the representation of structured knowledge in a particular domain should posses the following four properties:

(i) **Representational Adequacy**:- The ability to represent all kinds of knowledge that are needed in that domain.

(ii) **Inferential Adequacy** :- The ability to manipulate the represented structure and infer new structures.

(iii) **Inferential Efficiency**:- The ability to incorporate additional information into the knowledge structure that will aid the inference mechanisms.

(iv) **Acquisitional Efficiency** :- The ability to acquire new information easily, either by direct insertion or by program control.

The techniques that have been developed in AI systems to accomplish these objectives fall under two categories:

1. Declarative Methods:- In these knowledge is represented as static collection of facts which are manipulated by general procedures. Here the facts need to be stored only one and they can be used in any number of ways. Facts can be easily added to declarative systems without changing the general procedures.

2. Procedural Method:- In these knowledge is represented as procedures. Default reasoning and probabilistic reasoning are examples of procedural methods. In these, heuristic knowledge of "How to do things efficiently "can be easily represented.

In practice most of the knowledge representation employ a combination of both. Most of the knowledge representation structures have been developed to handle programs that handle natural language input. One of the reasons that knowledge structures are so important is that they provide a way to represent information about commonly occurring patterns of things . such descriptions are some times called schema. One definition of schema is

"Schema refers to an active organization of the past reactions, or of past experience, which must always be supposed to be operating in any well adapted organic response".

By using schemas, people as well as programs can exploit the fact that the real world is not random. There are several types of schemas that have proved useful in AI programs. They include

(i) **Frames**:- Used to describe a collection of attributes that a given object possesses (eg: description of a chair).

(ii) **Script**s:- Used to describe common sequence of events (eg:- a restaurant scene).

(iii) **Stereotypes** :- Used to described characteristics of people.

(iv) **Rule models**:- Used to describe common features shared among a set of rules in a production system.

Frames and scripts are used very extensively in a variety of AI programs. Before selecting any specific knowledge representation structure, the following issues have to be considered.

(i) The basis properties of objects , if any, which are common to every problem domain must be identified and handled appropriately.

(ii) The entire knowledge should be represented as a good set of primitives.

(iii) Mechanisms must be devised to access relevant parts in a large knowledge base.

2.3 Approaches to Knowledge Representation

Simple relational knowledge

The simplest way of storing facts is to use a relational method where each fact about a set of objects is set out systematically in columns. This representation gives little opportunity for inference, but it can be used as the knowledge basis for inference engines.

- Simple way to store facts.
- Each fact about a set of objects is set out systematically in columns (Fig. 7).
- Little opportunity for inference.
- Knowledge basis for inference engines.

Musician	Style	Instrument	Age
Miles Davis	Jazz	Trumpet	deceased
John Zorn	Avant Garde	Saxophone	35
Frank Zappa	Rock	Guitar	deceased
John Mclaughlin	Jazz	Guitar	47

Figure: Simple Relational Knowledge

We can ask things like:

- Who is dead?
- Who plays Jazz/Trumpet *etc.*?

This sort of representation is popular in database systems.

Inheritable knowledge

Relational knowledge is made up of objects consisting of

- attributes
- corresponding associated values.

We extend the base more by allowing inference mechanisms:

- Property inheritance
  - elements inherit values from being members of a class.
  - data must be organised into a hierarchy of classes (Fig. 8).

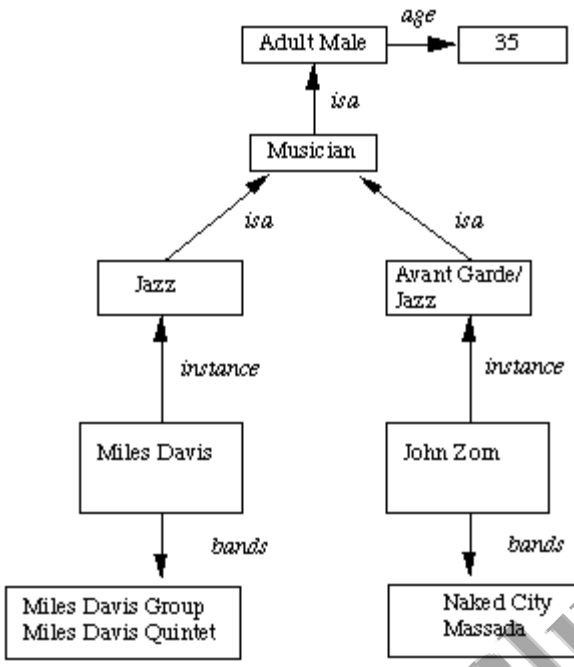


Fig. Property Inheritance Hierarchy

- Boxed nodes -- objects and values of attributes of objects.
- Values can be objects with attributes and so on.
- Arrows -- point from object to its value.
- This structure is known as a slot and filler structure, semantic network or a collection of frames.

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

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

Inferential Knowledge

Represent knowledge as *formal logic*:

All dogs have tails $\forall x$: $dog(x) \rightarrow hasatail(x)$ Advantages:

- A set of strict rules.
  - Can be used to derive more facts.
  - Truths of new statements can be verified.
  - Guaranteed correctness.
- Many inference procedures available to in implement standard rules of logic.
- Popular in AI systems. *e.g* Automated theorem proving.

Procedural Knowledge

Basic idea:

- Knowledge encoded in some procedures
  - small programs that know how to do specific things, how to proceed.

- *e.g* a parser in a natural language understander has the knowledge that a *noun phrase* may contain articles, adjectives and nouns. It is represented by calls to routines that know how to process articles, adjectives and nouns.

Advantages:

- *Heuristic* or domain specific knowledge can be represented.
- *Extended logical inferences*, such as default reasoning facilitated.
- *Side effects* of actions may be modelled. Some rules may become false in time. Keeping track of this in large systems may be tricky.

Disadvantages:

- Completeness -- not all cases may be represented.
- Consistency -- not all deductions may be correct.
  e.g If we know that *Fred is a bird* we might deduce that *Fred can fly*. Later we might discover that *Fred is an emu*.
- Modularity is sacrificed. Changes in knowledge base might have far-reaching effects.
- Cumbersome control information.

2.4 Issue in Knowledge Representation

Below are listed issues that should be raised when using a knowledge representation technique:

Important Attributes

-- Are there any attributes that occur in many different types of problem?
There are two *instance* and *isa* and each is important because each supports property inheritance.

Relationships

-- What about the relationship between the attributes of an object, such as, inverses, existence, techniques for reasoning about values and single valued attributes. We can consider an example of an inverse in
band(John Zorn,Naked City)
This can be treated as John Zorn plays in the band *Naked City* or John Zorn's band is *Naked City*.
Another representation is *band = Naked City*
band-members = John Zorn, Bill Frissell, Fred Frith, Joey Barron, . . .

Granularity

-- At what level should the knowledge be represented and what are the primitives.
Choosing the Granularity of Representation Primitives are fundamental concepts such as holding, seeing, playing and as English is a very rich language with over half a million words it is clear we will find difficulty in deciding upon which words to choose as our primitives in a series of situations.

If *Tom feeds a dog* then it could become:
feeds(tom, dog)
If *Tom gives the dog a bone* like:
gives(tom, dog,bone) Are these the same?
In any sense does giving an object food constitute feeding?
If *give*(*x*, *food*) $\rightarrow$ *feed(x)* then we are making progress.
But we need to add certain inferential rules.

In the famous program on relationships *Louise is Bill's cousin* How do we represent this? *louise = daughter (brother or sister (father or mother(bill)))* Suppose it is *Chris* then we do not know if it is *Chris* as a male or female and then *son* applies as well.
Clearly the separate levels of understanding require different levels of primitives and these need many rules to link together apparently similar primitives.
Obviously there is a potential storage problem and the underlying question must be what level of comprehension is needed.

2.5 Logic Knowledge Representation

We briefly mentioned how logic can be used to represent simple facts in the last lecture. Here we will highlight major principles involved in knowledge representation. In particular *predicate logic* will be met in other knowledge representation schemes and reasoning methods.
A more comprehensive treatment is given in the third year *Expert Systems* course. Symbols used
The following standard logic symbols we use in this course are:

For all	$\forall$
There exists	$\exists$
Implies	$\rightarrow$
Not	$\neg$
Or	$\vee$
And	$\wedge$

Let us now look at an example of how predicate logic is used to represent knowledge. There are other ways but this form is popular.

Propositional Logic.

Propositional logic is a special case of FOPL. It is simple to deal with and a decision procedure exists for it. We can easily represent real world facts as logical propositions. Propositions are elementary atomic sentences, written as formula or well formed formulas (wffs). Propositions may either be true or false . some examples of simple propositions are

it is raining

snow is white

people live on moon

Compound Propositions are formed from wffs using logical connectives " not, and or , if …then (implication) and if … and only if". For example :it is raining and wind is blowing" is a compound proposition.

As example of prepositional logic is given below

it is raining RAINING

it is hot HOT

it is windy WINDY

if it is raining then it is not hot
RAINING → ~ HOT
Propositional logic has it own limitations. Suppose we must to represent
Plato is a man
We might put it as MAN(PLATO)

Difficulty would arise if we want to represent the sentences like "All men are mortal" , "All elephants are gray " and " some fruits are rotten". Because such sentences need quantifications ie., way of representing "all" , "some" etc.

Predicate logic

An example
Consider the following:

- Prince is a mega star.
- Mega stars are rich.
- Rich people have fast cars.
- Fast cars consume a lot of petrol.

and try to draw the conclusion: *Prince's car consumes a lot of petrol*.
So we can translate *Prince is a mega star* into: *mega_star(prince)* and *Mega stars are rich* into: $\forall m$: *mega_star(m)* $\rightarrow$ *rich(m)*
Rich people have fast cars, the third axiom is more difficult:

- Is *cars* a relation and therefore *car(c,m)* says that case *c* is *m's* car. OR
- Is *cars* a function? So we may have *car_of(m)*.

Assume *cars* is a relation then axiom 3 may be written: $\forall c,m$: *car(c,m)* $\wedge$ *rich(m)* $\rightarrow$ *fast(c)*.
The fourth axiom is a general statement about *fast cars*. Let *consume(c)* mean that car *c* consumes a lot of petrol. Then we may write: $\forall c$: [*fast(c)* $\wedge \exists$ *m:car(c,m)* $\rightarrow$ *consume(c)*].

Is this enough? NO! -- Does prince have a car? We need the *car_of* function after all (and addition to *car*): $\forall c:car(car\_of(m),m)$. The result of applying *car_of* to *m* is *m*'s car. The final set of predicates is: $mega\_star(prince)$ $\forall m: mega\_star(m) \rightarrow rich(m)$ $\forall c:car(car\_of(m),m)$. $\forall c,m: car(c,m) \wedge rich(m) \rightarrow fast(c)$. $\forall c:[fast(c) \wedge \exists m:car(c,m) \rightarrow consume(c)]$. Given this we could conclude: $consume(car\_of(prince))$.

Isa and *instance* relationships

Two attributes *isa* and *instance* play an important role in many aspects of knowledge representation.

The reason for this is that they support *property inheritance*.

isa

-- used to show class inclusion, *e.g. isa(mega_star,rich).*

instance

-- used to show class membership, *e.g. instance(prince,mega_star).*

From the above it should be simple to see how to represent these in predicate logic.

Applications and extensions

- First order logic basically extends predicate calculus to allow:
  - functions -- return *objects* not just TRUE/FALSE.
  - equals predicate added.
- Problem solving and theorem proving -- large application areas.
- STRIPS robot planning system employs a first order logic system to enhance its means-ends analysis (GPS) planning. This amalgamation provided a very powerful heuristic search.
- Question answering systems.

Unification Algrithm.

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

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

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

(tryassassinate Marcus Caesar)

(tryassassinate Marcus (ruler of Rome))

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

(try assassinate Marcus Caesar)

(hate Marcus Caesar)

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

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

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

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

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

UNIFY (L1, L2)

1. if L1 or L2 is an atom part of same thing do

(a) if L1 or L2 are identical then return NIL

(b) else if L1 is a variable then do

(i) if L1 occurs in L2 then return F else return (L2/L1)

© else if L2 is a variable then do

(i) if L2 occurs in L1 then return F else return (L1/L2)

else return F.

2. If length (L!) is not equal to length (L2) then return F.

3. Set SUBST to NIL

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

4. For I = 1 to number of elements in L1 do

i) call UNIFY with the i th element of L1 and I'th element of L2, putting the result in S

ii) if S = F then return F

iii) if S is not equal to NIL then do

(A) apply S to the remainder of both L1 and L2
Asdf

2.6 RESOLUTION IN PREDICATE LOGIC

Two literals are contradictory if one can be unified with the negation of the other. For example man(x) and man (Himalayas) are contradictory since man(x) and man(Himalayas) can be unified. In predicate logic unification algorithm is used to locate pairs of literals that cancel out. It is important that if two instances of the same variable occur, then they must be given identical substitutions. The resolution algorithm for predicate logic as follows

Let f be a set of given statements and S is a statement to be proved.

1. Covert all the statements of F to clause form.

2. Negate S and convert the result to clause form. Add it to the set of clauses obtained in 1.

3. Repeat until either a contradiction is found or no progress can be made or a predetermined amount of effort has been expended.

a) Select two clauses. Call them parent clauses.

b) Resolve them together. The resolvent will be the disjunction of all of these literals of both clauses. If there is a pair of literals T1 and T2 such that one parent clause contains Ti and the other contains T2 and if T1 and T2 are unifiable, then neither t1 nor T2 should appear in the resolvent. Here Ti and T2 are called complimentary literals.

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

Clausal Form for Predicate Calculus !

In order to prove a formula in the predicate calculus by resolution,we

1.Negate the formula.

2.Put the negated formula into CNF, by doing the following:

i.Get rid of all $\rightarrow$operators.

ii.Push the $\neg$operators in as far as possible.

iii.Rename variables as necessary (see the step below).

iv.Move all of the quantifiers to the left (the outside) of the expression using the following rules (where **Q** is either $\forall$or $\exists$and **G** is a formula that does not contain **x**):

$\forall x\ R(x) \wedge \forall x\ S(x)$ becomes $\forall x\ (R(x) \wedge S(x))$.

$\exists x\ R(x) \vee \exists x\ S(x)$ becomes $\exists x\ (R(x) \vee S(x))$.

$Qx\ R(x) \wedge G$ becomes $Qx\ (R(x) \wedge G)$.

$Qx\ R(x) \vee G$ becomes $Qx\ (R(x) \vee G)$.

$Q_1x\ R(x) \wedge Q_2x\ S(x)$ becomes $Q_1x\ Q_2y\ (R(x) \wedge S(y))$.

$Q_1x\ R(x) \vee Q_2x\ S(x)$ becomes $Q_1x\ Q_2y\ (R(x) \vee S(y))$.

This leaves the formula in what is called *prenex form* which consists of a series of quantifiers followed by a quantifier-free formula, called the matrix.

v.Remove all quantifiers from the formula. First we remove the existentially quantified variables by using *Skolemization*. Each existentially quantified variable, say **x** is replaced by a function term which begins with a new, **n**-ary function symbol, say **f** where **n** is the number of universally quantified variables that occur before **x** is quantified in the formula. The arguments to the function term are precisely these variables. For example, if we have the formula

$$\forall x\ \forall y\ \exists z\ x + y < z,$$

then **z** would be replaced by a function term **f(x,y)** where **f** is a new function symbol. The result is:

$$\forall x\ \forall y\ x + y < f(x, y).$$

This new formula is satisfiable if and only if the original formula is satisfiable.

The new function symbol is called a Skolem function. If the existentially quantified variable has no preceding universally quantified variables, then the function is a 0-ary function and is often called a Skolem constant.

After removing all existential quantifiers, we simply drop all the universal quantifiers as we assume that any variable appearing in a formula is universally quantified.

vi.The remaining formula (the matrix) is put in CNF by moving any $\wedge$operators outside of any $\vee$operations.

3.Finally, the CNF formula is written in clausal format by writing each conjunct as a set of literals (a clause), and the whole formula as a set clauses (the clause set).

For example, if we begin with the proposition

$$\exists x\ \forall y\ y < x \rightarrow \forall y\ \exists x\ y < x$$

we have:

1.Negate the theorem:

$$\exists x\ \forall y\ y < x \wedge \exists y\ \forall x\ \neg(y < x)$$

i.Push the $\neg$operators in. No change.

ii).Rename variables if necessary:

$$\exists x\ \forall y\ y < x \wedge \exists u\ \forall v\ \neg(u < v)$$

iii)Move the quantifiers to the outside: First, we have

$$\exists x\ \exists u\ (\forall y\ y < x \wedge \forall v\ \neg(u < v))$$

Then we get

$$\exists x\ \exists u\ \forall y\ \forall v\ (y < x \wedge \neg(u < v))$$

iv)Remove the quantifiers, first by Skolemizing the existentially quantified variables. As these have no universally quantified variables to their left, they are replaced by Skolem constants:

$$\forall y\ \forall v\ (y < a \wedge \neg(b < v))$$

Drop the universal quantifiers:

$$y < a \wedge \neg(b < v)$$

v)Put the matrix into CNF. No change.

2.Write the formula in clausal form:

$$\{\{y < a\}, \{\neg(b < v)\}\}$$

Resolution !

Resolution is a rule of inference leading to a refutation theorem-proving technique for sentences in propositional logic and first-order logic. In other words, iteratively applying the resolution rule in a suitable way allows for telling whether a propositional formula is satisfiable and for proving that a first-order formula is unsatisfiable; this method may prove the satisfiability of a first-order satisfiable formula, but not always, as it is the case for all methods for first-order logic. Resolution was introduced by John Alan Robinson in 1965.

Resolution in propositional logic

The resolution rule in propositional logic is a single valid inference rule that produces a new clause implied by two clauses containing complementary literals. A literal is a propositional variable or the negation of a propositional variable. Two literals are said to be complements if one is the negation of the other (in the following, a_i is taken to be the complement to b_j). The resulting clause contains all the literals that do not have complements. Formally:

$$\frac{a_1 \vee \ldots \vee a_i \vee \ldots \vee a_n, \quad b_1 \vee \ldots \vee b_j \vee \ldots \vee b_m}{a_1 \vee \ldots \vee a_{i-1} \vee a_{i+1} \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_{j-1} \vee b_{j+1} \vee \ldots \vee b_m}$$

where

all as and bs are literals, a_i is the complement to b_j, and the dividing line stands for entails

The clause produced by the resolution rule is called the resolvent of the two input clauses.

When the two clauses contain more than one pair of complementary literals, the resolution rule can be applied (independently) for each such pair. However, only the pair of literals that are resolved upon can be removed: all other pair of literals remain in the resolvent clause.

A resolution technique, when coupled with a complete search algorithm, the resolution rule yields a sound and complete algorithm for deciding the satisfiability of a propositional formula, and, by extension, the validity of a sentence under a set of axioms.

This resolution technique uses proof by contradiction and is based on the fact that any sentence in propositional logic can be transformed into an equivalent sentence in conjunctive normal form. The steps are as follows:

1). All sentences in the knowledge base and the negation of the sentence to be proved (the conjecture) are conjunctively connected.

2). The resulting sentence is transformed into a conjunctive normal form with the conjuncts viewed as elements in a set, S, of clauses.

For example

$$(A_1 \vee A_2) \wedge (B_1 \vee B_2 \vee B_3) \wedge (C_1)$$

would give rise to a set

$$S = \{A_1 \vee A_2, B_1 \vee B_2 \vee B_3, C_1\}$$<="" p="">

3). The resolution rule is applied to all possible pairs of clauses that contain complementary literals. After each application of the resolution rule, the resulting sentence is simplified by removing repeated literals. If the sentence contains complementary literals, it is discarded (as a tautology). If not, and if it is not yet present in the clause set S, it is added to S, and is considered for further resolution inferences.

4). If after applying a resolution rule the empty clause is derived, the complete formula is unsatisfiable (or contradictory), and hence it can be concluded that the initial conjecture follows from the axioms.

5). If, on the other hand, the empty clause cannot be derived, and the resolution rule cannot be applied to derive any more new clauses, the conjecture is not a theorem of the original knowledge base.

One instance of this algorithm is the original Davis–Putnam algorithm that was later refined into the DPLL algorithm that removed the need for explicit representation of the resolvents.

This description of the resolution technique uses a set S as the underlying data-structure to represent resolution derivations. Lists, Trees and Directed Acyclic Graphs are other possible and common alternatives. Tree representations are more faithful to the fact that the resolution rule is binary. Together with a sequent notation for clauses, a tree representation also makes it clear to see how the resolution rule is related to a special case of the cut-rule, restricted to atomic cut-formulas. However, tree representations are not as compact as set or list representations, because they explicitly show redundant subderivations of clauses that are used more than once in the derivation of the empty clause. Graph representations can be as compact in the number of clauses as list representations and they also store structural information regarding which clauses were resolved to derive each resolvent.

Example

$$\frac{a \vee b, \quad \neg a \vee c}{b \vee c}$$

In English: if a or b is true, and a is false or c is true, then either b or c is true.

If a is true, then for the second premise to hold, c must be true. If a is false, then for the first premise to hold, b must be true.

So regardless of a, if both premises hold, then b or c is true.

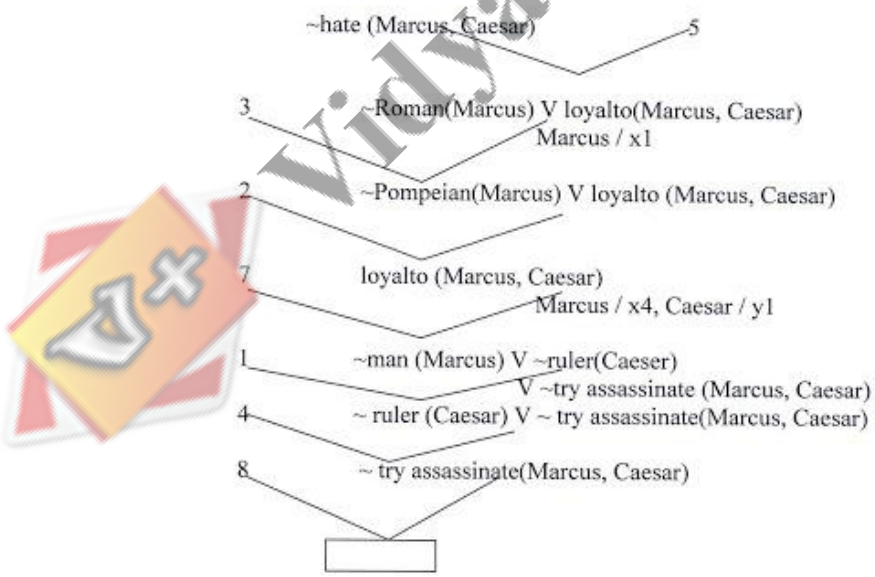
Using resolution to produce proof is illustrated in the following statements.

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

In order to use the above resolution roofs, they must be converted to clause form as given below:

1. man (Marcus)
2. Pompeian(Marcus)
3. ~ Pompeian(x1) V roman (x1)
   (from Pompeian (x1) ---> Roman (x1))
4. ruler (caesar)
5. ~Roman (x2) V loyalto (x2, Caesar) V hate (x2, Caesar)
6. loyalto (x3, f1(x3))
7. ~man(x4) V ~ ruler(y1) V ~ try assassinate(x4,y1) ~ loyalto (x4, y1))
8. try assassinate (Marcus, Caesar)

Suppose our goal is to answer the question of the assertion S, ie., hate (Marcus, Caesar) . The resolution procedure has been illustrated below.



The empty clause shows that ~hate (Marcus, Caesar) produces a contradiction or hate(Marcus, Caesar) will not produce a contradiction with the known statements.

Rajalakshmi Institute of Technology
Department of Information Technology

CS6659 - Artificial Intelligence

UNIT III - KNOWLEDGE INFERENCE

Knowledge representation - Production based system, Frame based system. Inference – Backward chaining, Forward chaining, Rule value approach, Fuzzy reasoning – Certainty factors, Bayesian Theory-Bayesian Network-Dempster – Shafer theory.

KNOWLEDGE REPRESENTATION

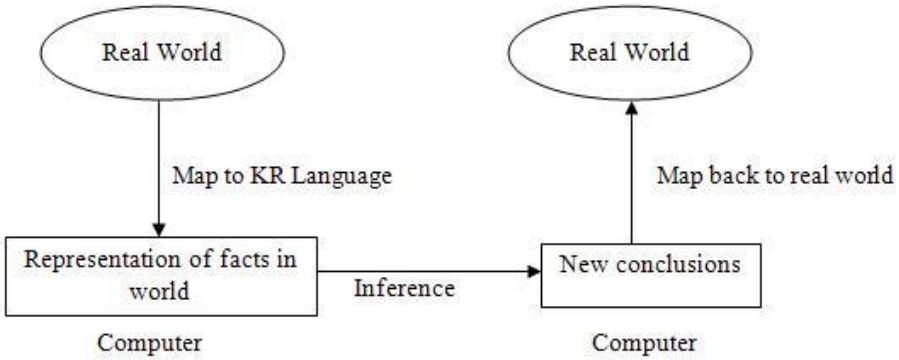
Knowledge representation is probably, the most important ingredient for developing an AI. A representation is a layer between information accessible from outside world and high level thinking processes. Without knowledge representation it is impossible to identify what thinking processes are, mainly because representation itself is a substratum for a thought.

The need of knowledge representation was felt as early as the idea to develop intelligent systems. With the hope that readers are well conversant with the fact by now, that intelligent requires possession of knowledge and that knowledge is acquired by us by various means and stored in the memory using some representation techniques. Putting in another way, knowledge representation is one of the many critical aspects, which are required for making a computer behave intelligently. Knowledge representation refers to the data structures techniques and organizing notations that are used in AI. These include semantic networks, frames, logic, production rules and conceptual graphs.

Syntax and semantics for Knowledge Representation

Knowledge representation languages should have precise syntax and semantics. You must know exactly what an expression means in terms of objects in the real world. Suppose we have decided that “red 1” refers to a dark red colour, “car1” is my car, car2 is another. Syntax of language will tell you which of the following is legal: red1 (car1), red1 car1, car1 (red1), red1 (car1 & car2)?

Semantics of language tell you exactly what an expression means: for example, Pred (Arg) means that the property referred to by Pred applies to the object referred to by Arg. E.g., properly “dark red” appli es to my car.



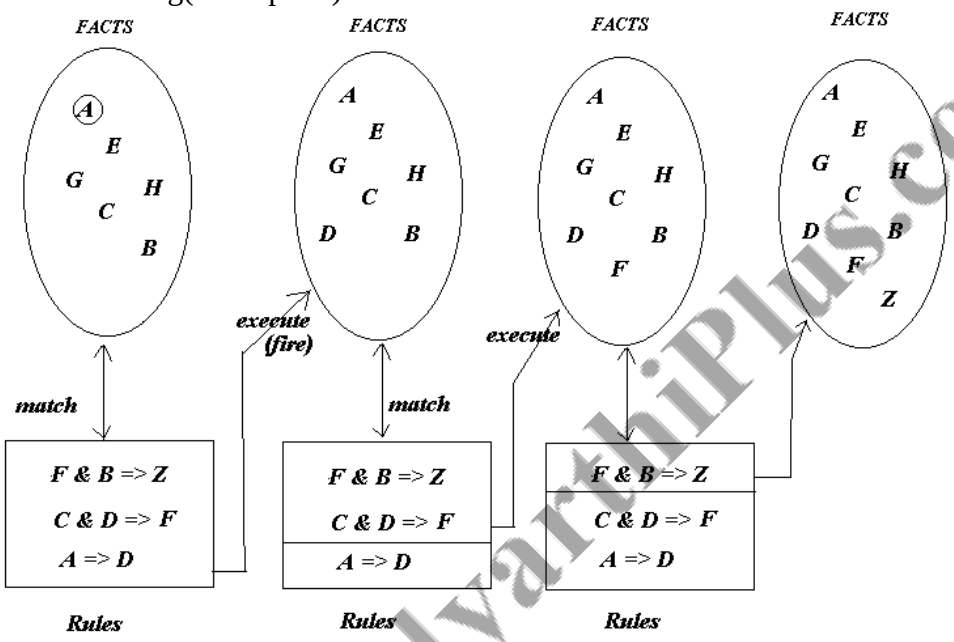
Production Based System

REASONING ABOUT UNCERTAINTY

An inference engine should be able to handle incomplete information. The degree of certainty is represented as a number of attached to a fact (certainty factor). There are three inferencing methods. These are Forward, Backward and Mixed Chaining.

FORWARD CHAINING

Forward Chaining(Example 1)



Problem: Does situation Z exists or not ?

The first rule that fires is A->D because A is already in the database. Next we infer D. Existence of C and D causes second rule to fire and as a consequence F is inferred and placed in the database. This in turn, causes the third rule F?B->Z to fire, placing Z in the database.

This technique is called forward chaining.

A very simple Forward chaining Algorithm

Given m facts $F_1,F_2,....,F_m$? N RULES

$R_1,R_2,.....R_n$

repeat

for i ?- 1 to n do

if one or more current facts match the antecedent of Ri then

1) add the new fact(s) define by the consequent

2) flag the rule that has been fired

3) increase m

until no new facts have been produced.

Forward Chaining (Example 2)

Rule 1
IF the car overheats , THEN the car will stall.
Rule 2
IF the car stalls
THEN it will cost me money
AND I will be late getting home

Now, the question is
How do you arrive at conclusion that this situation will cost money and cause you to be late ?
The condition that triggers the chain of events is the car overheating

BACKWARD CHAINING

Backward Chaining (Example 1)

Rule 1
IF the car is not tuned AND the battery is weak
THEN not enough current will reach the starter.
Rule 2
IF not enough current reaches the starter
THEN the car will not start.

Given facts:
The car is not tuned
The battery is weak.
Now, the question is
How would you arrive at the conditions that have resulted in the car failing to start?

Backward Chaining(Example 2)

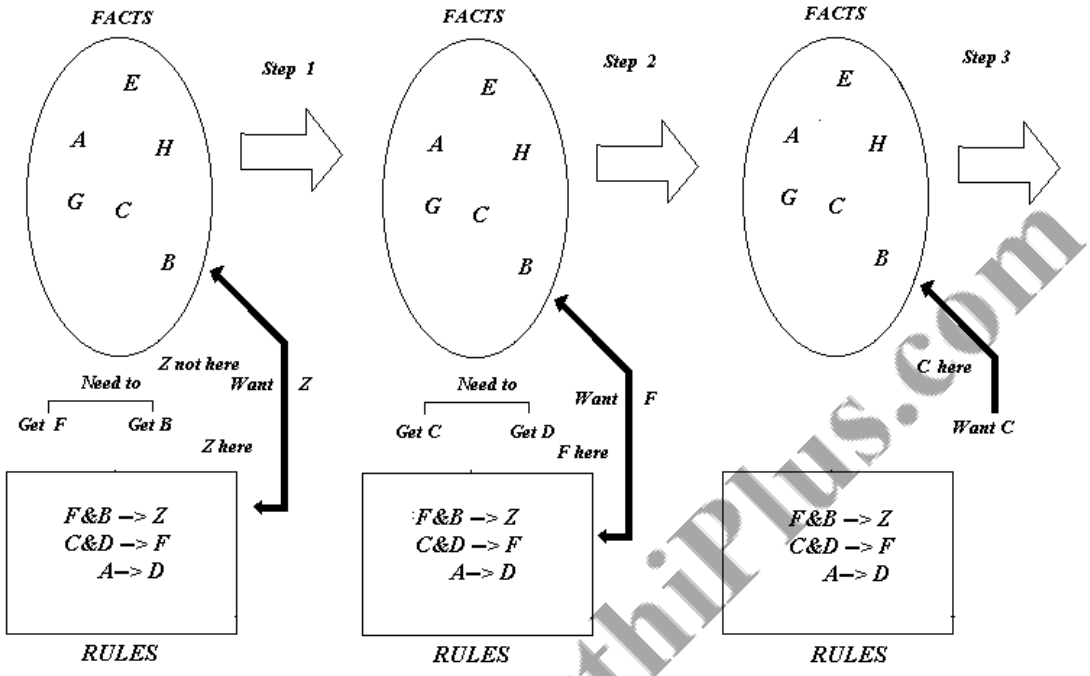
In such a situation backward chaining might be more cost-effective.With this inference method the system starts with what it wants to prove, e.g.,that situation Z exists, and only executes rules that are relavent to establishing it.Figure following shows how bacward chaining would work using the rules from the forward chaining example.

In step 1 the system is told to establish (if it can) that situation Z exists,It first checks the data base for Z,and when that fails, searches for rules that conclude Z,i.e., have Z on the right side of the arrow.It finds the rule F?B->Z, and decides that it must establish F and B in order to conclude Z.

In step 2 the system tries to establish F, first checking the data base and then finding a rule that concludes F. From this rule, C?D->F, the system decides it must establish C and D to conclude F.

In steps 3 through 5 the system finds C in the data base but decides it must establish A before it can conclude D. It then finds A in the data base.

In steps 6 through 8 the system executes the third rule to establish D, then executes the second rule to establish the original goal, Z.The infenece chain created here is identical to the one created by forward chaining. The difference in two approaches hinges on the method in which data and rules are searched.



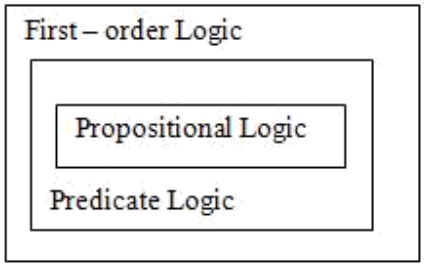
Types of Knowledge Representation

Knowledge can be represented in different ways. The structuring of knowledge and how designers might view it, as well as the type of structures used internally are considered. Different knowledge representation techniques are

a. Logic
b. Semantic Network
c. Frame
d. Conceptual Graphs
e. Conceptual Dependency
f. Script

Logic

A logic is a formal language, with precisely defined syntax and semantics, which supports sound inference. Different logics exist, which allow you to represent different kinds of things, and which allow more or less efficient inference. The logic may be different types like propositional logic, predicate logic, temporal logic, description logic etc. But representing something in logic may not be very natural and inferences may not be efficient.

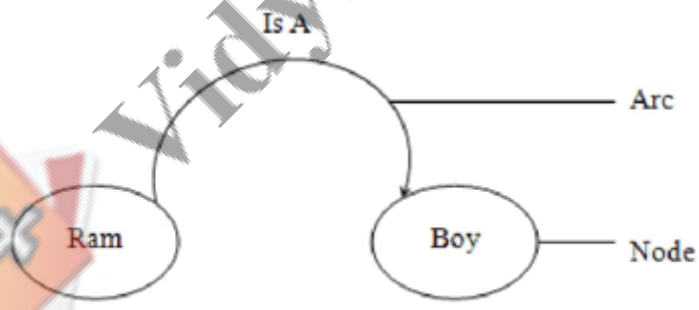


Semantic Network

A semantic network is a graphical knowledge representation technique. This knowledge representation system is primarily on network structure. The semantic networks were basically developed to model human memory. A semantic net consists of nodes connected by arcs. The arcs are defined in a variety of ways, depending upon the kind of knowledge being represented.

The main idea behind semantic net is that the meaning of a concept comes, from the ways in which it is connected to other concepts. The semantic network consists of different nodes and arcs. Each node should contain the information about objects and each arc should contain the relationship between objects. Semantic nets are used to find relationships among objects by spreading activation about from each of two nodes and seeing where the activation met this process is called intersection search.

For example: Ram is a boy.

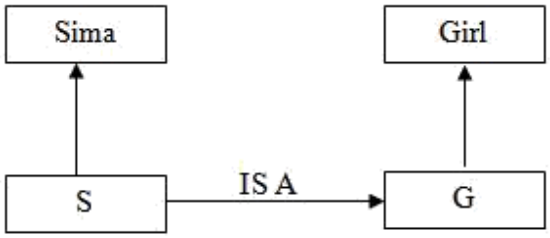


Semantic network by using Instances

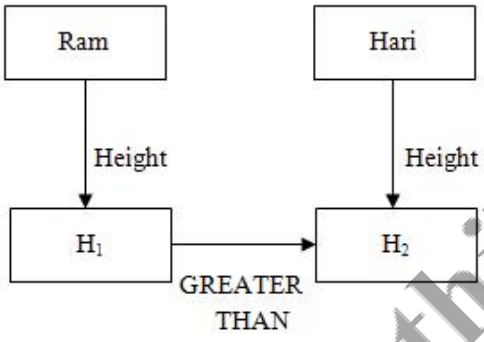
The semantic network based knowledge representation mechanism is useful where an object or concept is associated with many attributes and where relationships between objects are important. Semantic nets have also been used in natural language research to represent complex sentences expressed in English. The semantic representation is useful because it provides a standard way of analyzing the meaning of sentence. It is a natural way to represent relationships that would appear as ground instances of binary predicates in predicate logic. In this case we can create one instance of each object. In instance based semantic net representations some keywords are used like: IS A, INSTANCE, AGENT, HAS-PARTS etc.

Consider the following examples:

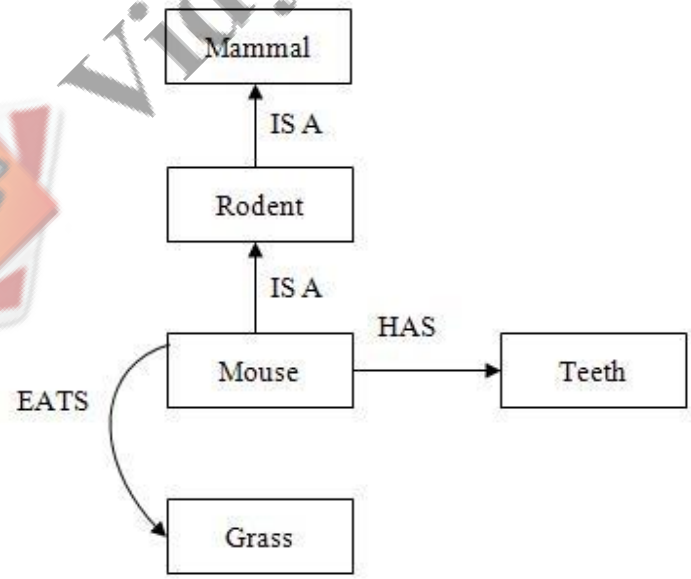
1. Suppose we have to represent the sentence “Sima is a girl”.



2. Ram is taller than Hari
   It can also be represented as



3. “Mouse is a Rodent and Rodent is a mammal. Mouse ha s teeth and etas grass”. Check whether the sentence mammal has teeth is valid or not.]



Partitioned Semantic Network

Some complex sentences are there which cannot be represented by simple semantic nets and for this we

have to follow the technique partitioned semantic networks. Partitioned semantic net allow for

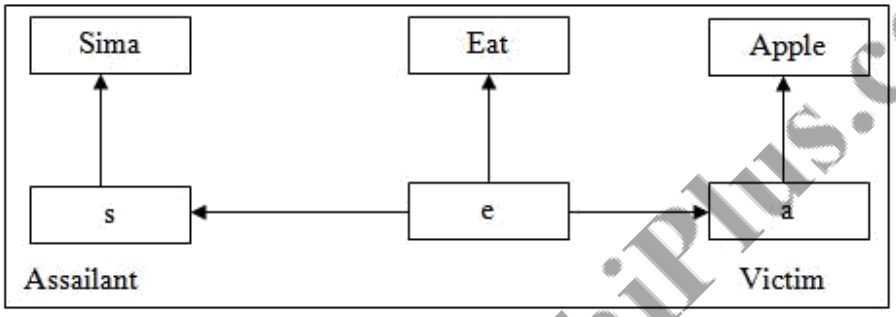
1. Propositions to be made without commitment to truth.
2. Expressions to be quantified.

In partitioned semantic network, the network is broken into spaces which consist of groups of nodes and arcs and regard each space as a node.

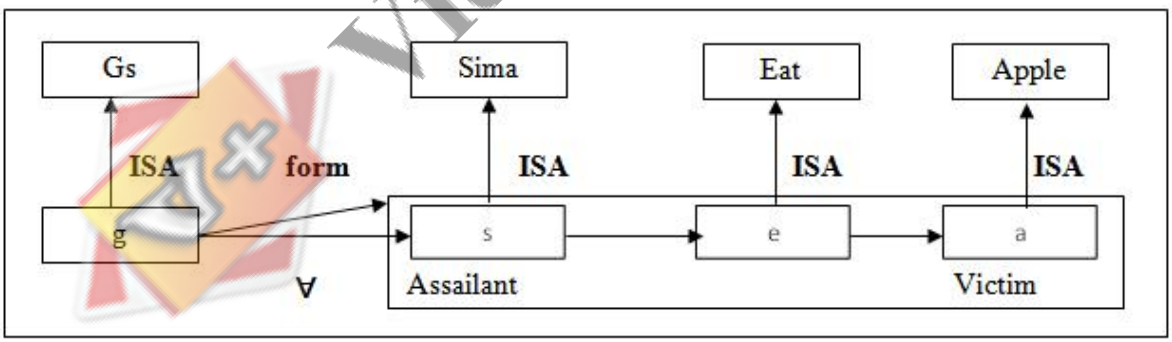
Let us consider few examples.

Draw the partitioned semantic network structure for the followings:

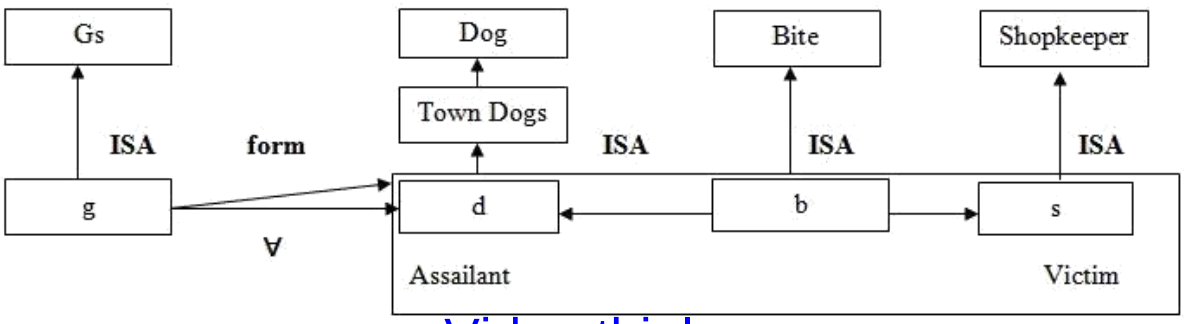
a) Sima is eating an apple.



b) All Sima are eating an apple.



c. Every dog in town has bitten a shopkeeper.



FRAME

A frame is a collection of attributes and associated values that describe some entity in the world. Frames are general record like structures which consist of a collection of slots and slot values. The slots may be of any size and type. Slots typically have names and values or subfields called facets. Facets may also have names and any number of values. A frame may have any number of slots, a slot may have any number of facets, each with any number of values. A slot contains information such as attribute value pairs, default values, condition for filling a slot, pointers to other related frames and procedures that are activated when needed for different purposes. Sometimes a frame describes an entity in some absolute sense, sometimes it represents the entity from a particular point of view. A single frame taken alone is rarely useful. We build frame systems out of collection of frames that are connected to each other by virtue of the fact that the value of an attribute of one frame may be another frame. Each frame should start with an open parenthesis and closed with a closed parenthesis.

Syntax of a frame

(<frame name>
(<slot1> (<facet1> <value 1>.....<value n_1>)
(<facet2> <value1>.....<value n_2>)
.
.
.
.
.
(<facet n> <value1>..... <value n_n>))
(<slot 2> (<facet1> <value 1>.....<value n_1>)
(<facet2><value2>.....<value n_2>)
.
.
))

Let us consider the below examples.

1) Create a frame of the person Ram who is a doctor. He is of 40. His wife name is Sita. They have two children Babu and Gita. They live in 100 kps street in the city of Delhi in India. The zip code is 756005.

(Ram

(PROFESSION (VALUE Doctor))

(AGE (VALUE 40))

(WIFE (VALUE Sita))

(CHILDREN (VALUE Bubu, Gita))

(ADDRESS

(STREET (VALUE 100 kps))

(CITY(VALUE Delhi))

(COUNTRY(VALUE India))

(ZIP (VALUE 756005))))

SCRIPT

It is an another knowledge representation technique. Scripts are frame like structures used to represent commonly occurring experiences such as going to restaurant, visiting a doctor. A script is a structure that describes a stereotyped sequence of events in a particular context. A script consist of a set of slots. Associated with each slot may be some information about what kinds of values it may contain as well as a default value to be used if no other information is available. Scripts are useful because in the real world, there are no patterns to the occurrence of events. These patterns arise because of clausal relationships between events. The events described in a script form a giant casual chain. The beginning of the chain is the set of entry conditions which enable the first events of the script to occur. The end of the chain is the set of results which may enable later events to occur. The headers of a script can all serve as indicators that the script should be activated.

Once a script has been activated, there are a variety of ways in which it can be useful in interpreting a particular situation. A script has the ability to predict events that has not explicitly been observed. An important use of scripts is to provide a way of building a single coherent interpretation from a collection of observation. Scripts are less general structures than are frames and so are not suitable for representing all kinds of knowledge. Scripts are very useful for representing the specific kinds of knowledge for which they were designed.

A script has various components like:

1) **Entry condition**: It must be true before the events described in the script can occur. E.g. in a restaurant script the entry condition must be the customer should be hungry and the customer has money.
2) **Tracks:** It specifies particular position of the script e.g. In a supermarket script the tracks may be cloth gallery, cosmetics gallery etc.
3) **Result:** It must be satisfied or true after the events described in the script have occurred. e.g.
   In a restaurant script the result must be true if the customer is pleased.

   The customer has less money.
4) **Probs:** It describes the inactive or dead participants in the script e.g. In a supermarket script, the probes may be clothes, sticks, doors, tables, bills etc.
5) **Roles:** It specifies the various stages of the script. E.g. In a restaurant script the scenes may be entering, ordering etc.
   Now let us look on a movie script description according to the above component.

   a) Script name : Movie
   b) Track : CINEMA HALL

c) Roles : Customer(c), Ticket seller(TS), Ticket Checker(TC), Snacks Sellers (SS)

d) Probes : Ticket, snacks, chair, money, Ticket, chart

e) Entry condition : The customer has money

The customer has interest to watch movie.

6) **Scenes:**

a. SCENE-1 (Entering into the cinema hall)

C PTRANS C into the cinema hall

C ATTEND eyes towards the ticket counter C PTRANS C towards the ticket counters C ATTEND eyes to the ticket chart

C MBUILD to take which class ticket C MTRANS TS for ticket

C ATRANS money to TS

TS ATRANS ticket to C

b. SCENE-2 (Entering into the main ticket check gate)

C PTRANS C into the queue of the gate C ATRANS ticket to TC

TC ATTEND eyes onto the ticket

TC MBUILD to give permission to C for entering into the hall

TC ATRANS ticket to C

C PTRANS C into the picture hall.

Example 2: Write a script of visiting a doctor in a hospital

1) SCRIPT_NAME : Visiting a doctor
2) TRACKS : Ent specialist
3) ROLES : Attendant (A), Nurse(N), Chemist (C), Gatekeeper(G), Counter clerk(CC), Receptionist(R), Patient(P), Ent specialist Doctor (D), Medicine Seller (M).

4) PROBES : Money, Prescription, Medicine, Sitting chair, Doctor's table, Thermometer, Stetho scope, writing pad, pen, torch, stature.

5) ENTRY CONDITION: The patient need consultation. Doctor's visiting time on.

6) SCENES:

a. SCENE-1 (Entering into the hospital)

P PTRANS P into hospital

P ATTEND eyes towards ENT department

P PTRANS P into ENT department

P PTRANS P towards the sitting chair **b. SCENE-2 (Entering into the Doctor's Room)**

P PTRANS P into doctor's room

P MTRANS P about the diseases

P SPEAK D about the disease

D MTRANS P for blood test, urine test

D ATRANS prescription to P

P PTRANS prescription to P.

P PTRANS P for blood and urine test **c. SCENE-3 (Entering into the Test Lab)**

P PTRANS P into the test room

P ATRANS blood sample at collection room

P ATRANS urine sample at collection room

P ATRANS the examination reports

d. SCENE-4 (Entering to the Doctor's room with Test reports)

P ATRANS the report to D

D ATTEND eyes into the report

D MBUILD to give the medicines

7) RESULT:

The patient has less money and Patient has prescription and medicine.

Advantages and Disadvantages of Different Knowledge Representation

Sl. No.	Scheme	Advantages	Disadvantages
1	Production rules	• Simple syntax • Easy to understand • Simple interpreter • Highly Modular • Easy to add or modify	• Hard to follow Hierarchies • Inefficient for large systems • Poor at representing structured descriptive knowledge.
2	Semantic	• Easy to follow hierarchy • Easy to trace associations • Flexible	• Meaning attached to nodes might be ambiguous • Exception handling is difficult • Difficult to program
3	Frame	• Expressive Power • Easy to set up slots for new properties and relations • Easy to create specialized	• Difficult to program • Difficult for inference • Lack of inexpensive software
4	Script	• Ability to predict events • A single coherent interpretation may be build up from a collection of observations	• Less general than frames • May not be suitable to represent all kinds of knowledge
5	Formal Logic	• Facts asserted independently of use • Assurance that only valid consequence are asserted • Completeness	• Separation of representation and processing • Inefficient with large data sets • Very slow with large knowledge bases

Conceptual Graphs

A conceptual graph (CG) is a graph representation for logic based on the semantic networks of artificial intelligence.

A conceptual graph consists of concept nodes and relation nodes.

- The concept nodes represent entities, attributes, states, and events
- The relation nodes show how the concepts are interconnected

Conceptual Graphs are finite, connected, bipartite graphs.

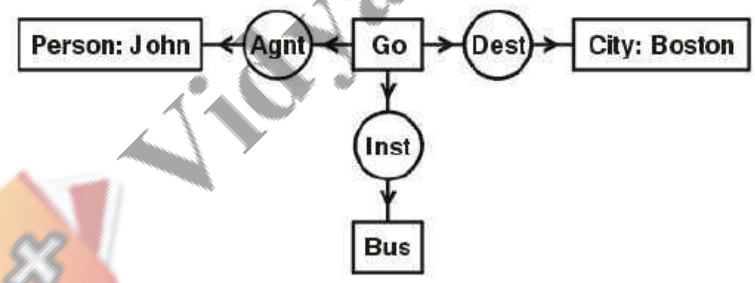
Finite: because any graph (in 'human brain' or 'computer storage') can only have a finite number of concepts and conceptual relations.

Connected: because two parts that are not connected would simply be called two conceptual graphs.

Bipartite: because there are two different kinds of nodes: concepts and conceptual relations, and every arc links a node of one kind to a node of another kind

Example

Following CG display form for John is going to Boston by bus.



The conceptual graph in Figure represents a typed or sorted version of logic. Each of the four concepts has a type label, which represents the type of entity the concept refers to: Person, Go, Boston, or Bus. Two of the concepts have names, which identify the referent: John or Boston. Each of the three conceptual relations has a type label that represents the type of relation: agent (Agnt), destination (Dest), or instrument (Inst). The CG as a whole indicates that the person John is the agent of some instance of going, the city Boston is the destination, and a bus is the instrument. Figure 1 can be translated to the following formula:

$$(\exists x)(\exists y)(\text{Go}(x) \wedge \text{Person}(\text{John}) \wedge \text{City}(\text{Boston}) \wedge \text{Bus}(y) \wedge \text{Agnt}(x,\text{John}) \wedge \text{Dest}(x,\text{Boston}) \wedge \text{Inst}(x,y))$$

As this translation shows, the only logical operators used in Figure are conjunction and the existential quantifier. Those two operators are the most common in translations from natural languages, and many of the early semantic networks could not represent any others.

Inference Rules

Complex deductive arguments can be judged valid or invalid based on whether or not the steps in that argument follow the nine basic rules of inference. These rules of inference are all relatively simple, although when presented in formal terms they can look overly complex.

Conjunction:

1. P
2. Q
3. Therefore, P and Q.

1. It is raining in New York.
2. It is raining in Boston
3. Therefore, it is raining in both New York and Boston

Simplification

1. P and Q.
2. Therefore, P.

1. It is raining in both New York and Boston.
2. Therefore, it is raining in New York.

Addition

1. P
2. Therefore, P or Q.

1. It is raining
2. Therefore, either either it is raining or the sun is shining.

Absorption

1. If P, then Q.
2. Therfore, If P then P and Q.

1. If it is raining, then I will get wet.
2. Therefore, if it is raining, then it is raining and I will get wet.

Modus Ponens

1. If P then Q.
2. P.
3. Therefore, Q.

1. If it is raining, then I will get wet.
2. It is raining.
3. Therefore, I will get wet.

Modus Tollens

1. If P then Q.
2. Not Q. (~Q).
3. Therefore, not P (~P).

1. If it had rained this morning, I would have gotten wet.
2. I did not get wet.
3. Therefore, it did not rain this morning.

Hypothetical Syllogism

1. If P then Q.
2. If Q then R.
3. Therefore, if P then R.

1. If it rains, then I will get wet.
2. If I get wet, then my shirt will be ruined.
3. If it rains, then my shirt will be ruined.

Disjunctive Syllogism

1. Either P or Q.
2. Not P (~P).
3. Therefore, Q.

1. Either it rained or I took a cab to the movies.
2. It did not rain.
3. Therefore, I took a cab to the movies.

Constructive Dilemma

1. (If P then Q) and (If R then S).
2. P or R.
3. Therefore, Q or S.

1. If it rains, then I will get wet and if it is sunny, then I will be dry.
2. Either it will rain or it will be sunny.
3. Therefore, either I will get wet or I will be dry.

The above rules of inference, when combined with the rules of replacement, mean that propositional calculus is "complete." Propositional calculus is simply another name for formal logic.

Managing Uncertainty in Expert Systems

Sources of uncertainty in Expert System

- Weak implication
- Imprecise language
- Unknown data
- Difficulty in combining the views of different experts

Uncertainty in AI

- Information is partial
- Information is not fully reliable
- Representation language is inherently imprecise
- Information comes from multiple sources and it is conflicting
- Information is approximate
- Non-absolute cause-effect relationship exist

Representing uncertain information in Expert System

- Probabilistic
- Certainty factors
- Theory of evidence
- Fuzzy logic
- Neural Network
- GA
- Rough set

Nonmonotonic logic and Reasoning with Beliefs

A non-monotonic logic is a formal logic whose consequence relation is not monotonic. Most studied formal logics have a monotonic consequence relation, meaning that adding a formula to a theory never produces a reduction of its set of consequences. Intuitively, monotonicity indicates that learning a new piece of knowledge cannot reduce the set of what is known. A monotonic logic cannot handle various reasoning tasks such as reasoning by default (consequences may be derived only because of lack of evidence of the contrary), abductive reasoning (consequences are only deduced as most likely explanations) and some important approaches to reasoning about

knowledge (the ignorance of a consequence must be retracted when the consequence becomes known) and similarly belief revision (new knowledge may contradict old beliefs).

Default reasoning

An example of a default assumption is that the typical bird flies. As a result, if a given animal is known to be a bird, and nothing else is known, it can be assumed to be able to fly. The default assumption must however be retracted if it is later learned that the considered animal is a penguin. This example shows that a logic that models default reasoning should not be monotonic. Logics formalizing default reasoning can be roughly divided in two categories: logics able to deal with arbitrary default assumptions (default logic, defeasible logic/defeasible reasoning/argument (logic), and answer set programming) and logics that formalize the specific default assumption that facts that are not known to be true can be assumed false by default (closed world assumption and circumscription).

Abductive reasoning

Abductive reasoning is the process of deriving the most likely explanations of the known facts. An abductive logic should not be monotonic because the most likely explanations are not necessarily correct. For example, the most likely explanation for seeing wet grass is that it rained; however, this explanation has to be retracted when learning that the real cause of the grass being wet was a sprinkler. Since the old explanation (it rained) is retracted because of the addition of a piece of knowledge (a sprinkler was active), any logic that models explanations is non-monotonic.

Reasoning about knowledge

If a logic includes formulae that mean that something is not known, this logic should not be monotonic. Indeed, learning something that was previously not known leads to the removal of the formula specifying that this piece of knowledge is not known. This second change (a removal caused by an addition) violates the condition of monotonicity. A logic for reasoning about knowledge is the autoepistemic logic.

Belief revision

Belief revision is the process of changing beliefs to accommodate a new belief that might be inconsistent with the old ones. In the assumption that the new belief is correct, some of the old ones have to be retracted in order to maintain consistency. This retraction in response to an addition of a new belief makes any logic for belief revision to be non-monotonic. The belief revision approach is alternative to paraconsistent logics, which tolerate inconsistency rather than attempting to remove it.

What makes belief revision non-trivial is that several different ways for performing this operation may be possible. For example, if the current knowledge includes the three facts “A is true”, “B is true” and “if A and B are true then C is true”, the introduction of the new information “C is false” can be done preserving consistency only by removing at least one of the three facts.

In this case, there are at least three different ways for performing revision. In general, there may be several different ways for changing knowledge.

Bayesian Probability Theory

Bayesian probability is one of the most popular interpretations of the concept of probability. The Bayesian interpretation of probability can be seen as an extension of logic that enables reasoning with uncertain statements. To evaluate the probability of a hypothesis, the Bayesian probabilist specifies some prior probability, which is then updated in the light of new relevant data. The Bayesian interpretation provides a standard set of procedures and formulae to perform this calculation.

Bayesian probability interprets the concept of probability as "a measure of a state of knowledge", in contrast to interpreting it as a frequency or a physical property of a system. Its name is derived from the 18th century statistician Thomas Bayes, who pioneered some of the concepts. Broadly speaking, there are two views on Bayesian probability that interpret the state of knowledge concept in different ways. According to the objectivist view, the rules of Bayesian statistics can be justified by requirements of rationality and consistency and interpreted as an extension of logic. According to the subjectivist view, the state of knowledge measures a "personal belief". Many modern machine learning methods are based on objectivist Bayesian principles. One of the crucial features of the Bayesian view is that a probability is assigned to a hypothesis, whereas under the frequentist view, a hypothesis is typically rejected or not rejected without directly assigning a probability.

The probability of a hypothesis given the data (the posterior) is proportional to the product of the likelihood times the prior probability (often just called the prior). The likelihood brings in the effect of the data, while the prior specifies the belief in the hypothesis before the data was observed.

More formally, Bayesian inference uses Bayes' formula for conditional probability:

$$P(H|D) = \frac{P(D|H)\ P(H)}{P(D)}$$

where

H is a hypothesis, and D is the data.

P(H) is the prior probability of H: the probability that H is correct before the data D was seen.

P(D | H) is the conditional probability of seeing the data D given that the hypothesis H is true. P(D | H) is called the likelihood.

P(D) is the marginal probability of D.

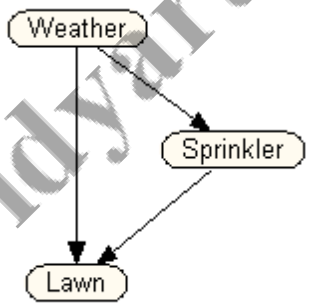
P(H | D) is the posterior probability: the probability that the hypothesis is true, given the data and the previous state of belief about the hypothesis.

What is a Bayes net?

A Bayes net is a model. It reflects the states of some part of a world that is being modeled and it describes how those states are related by probabilities. The model might be of your house, or your car, your body, your community, an ecosystem, a stock-market, etc. Absolutely anything can be modeled by a Bayes net. All the possible states of the model represent all the possible worlds that can exist, that is, all the possible ways that the parts or states can be configured. The car engine can be running normally or giving trouble. It's tires can be inflated or flat. Your body can be sick or healthy, and so on.

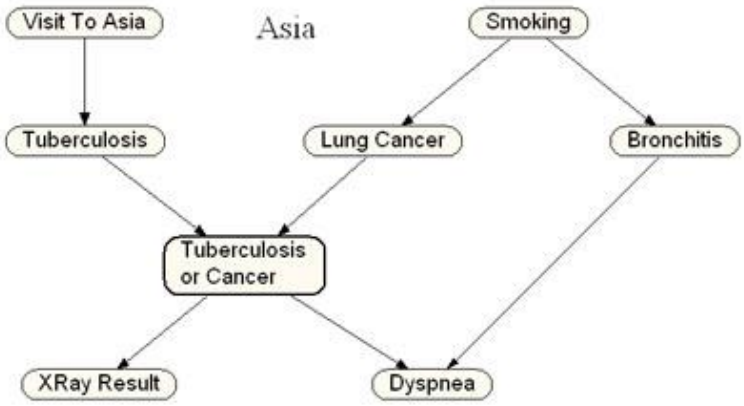
So where do the probabilities come in? Well, typically some states will tend to occur more frequently when other states are present. Thus, if you are sick, the chances of a runny nose are higher. If it is cloudy, the chances of rain are higher, and so on.

Here is a simple Bayes net that illustrates these concepts. In this simple world, let us say the weather can have three states: sunny, cloudy, or rainy, also that the grass can be wet or dry, and that the sprinkler can be on or off. Now there are some causal links in this world. If it is rainy, then it will make the grass wet directly. But if it is sunny for a long time, that too can make the grass wet, indirectly, by causing us to turn on the sprinkler.



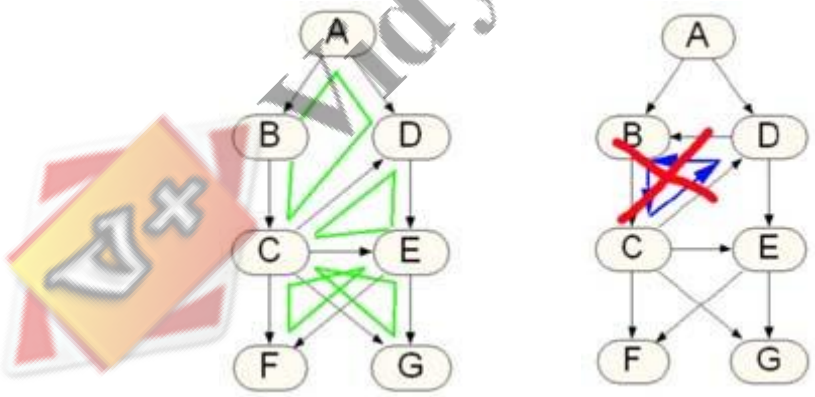
When actual probabilities are entered into this net that reflect the reality of real weather, lawn, and sprinkler-use-behavior, such a net can be made to answer a number of useful questions, like, "if the lawn is wet, what are the chances it was caused by rain or by the sprinkler", and "if the chance of rain increases, how does that affect my having to budget time for watering the lawn".

Here is another simple Bayes net called **Asia**. It is an example which is popular for introducing Bayes nets and is from Lauritzen&Spiegelhalter88. Note, it is for example purposes only, and should not be used for real decision making.



It is a simplified version of a network that could be used to diagnose patients arriving at a clinic. Each node in the network corresponds to some condition of the patient, for example, "Visit to Asia" indicates whether the patient recently visited Asia. The arrows (also called links) between any two nodes indicate that there are probability relationships that are know to exist between the states of those two nodes. Thus, smoking increases the chances of getting lung cancer and of getting bronchitis. Both lung cancer and bronchitis increase the chances of getting dyspnea (shortness of breath). Both lung cancer and tuberculosis, but not usually bronchitis, can cause an abnormal lung x-ray. And so on.

In a Bayes net, the links may form loops, but they may not form cycles. This is not an expressive limitation; it does not limit the modeling power of these nets. It only means we must be more careful in building our nets. In the left diagram below, there are numerous loops. These are fine. In the right diagram, the addition of the link from D to B creates a cycle, which is not permitted.



A valid Bayes net Not a Bayes net

The key advantage of not allowing cycles it that it makes possible very fast update algorithms, since there is no way for probabilistic influence to "cycle around" indefinitely.

To diagnose a patient, values could be entered for some of nodes when they are known. This would allow us to re-calculate the probabilities for all the other nodes. Thus if we take a chest x-ray and the x-ray is abnormal, then the chances of the patient having TB or lung-cancer rise. If

we further learn that our patient visited Asia, then the chances that they have tuberculosis would rise further, and of lung-cancer would drop (since the X-ray is now better explained by the presence of TB than of lung-cancer). We will see how this is done in a later section.

Dempster-Shafer Theory

The Dempster-Shafer theory, also known as the theory of belief functions, is a generalization of the Bayesian theory of subjective probability. Whereas the Bayesian theory requires probabilities for each question of interest, belief functions allow us to base degrees of belief for one question on probabilities for a related question. These degrees of belief may or may not have the mathematical properties of probabilities; how much they differ from probabilities will depend on how closely the two questions are related.

The Dempster-Shafer theory owes its name to work by A. P. Dempster (1968) and Glenn Shafer (1976), but the kind of reasoning the theory uses can be found as far back as the seventeenth century. The theory came to the attention of AI researchers in the early 1980s, when they were trying to adapt probability theory to expert systems. Dempster-Shafer degrees of belief resemble the certainty factors in MYCIN, and this resemblance suggested that they might combine the rigor of probability theory with the flexibility of rule-based systems. Subsequent work has made clear that the management of uncertainty inherently requires more structure than is available in simple rule-based systems, but the Dempster-Shafer theory remains attractive because of its relative flexibility.

The Dempster-Shafer theory is based on two ideas: the idea of obtaining degrees of belief for one question from subjective probabilities for a related question, and Dempster's rule for combining such degrees of belief when they are based on independent items of evidence.

To illustrate the idea of obtaining degrees of belief for one question from subjective probabilities for another, suppose I have subjective probabilities for the reliability of my friend Jon. My probability that he is reliable is 0.9, and my probability that he is unreliable is 0.1. Suppose he tells me a limb fell on my car. This statement, which must true if she is reliable, is not necessarily false if she is unreliable. So his testimony alone justifies a 0.9 degree of belief that a limb fell on my car, but only a zero degree of belief (not a 0.1 degree of belief) that no limb fell on my car. This zero does not mean that I am sure that no limb fell on my car, as a zero probability would; it merely means that jon's testimony gives me no reason to believe that no limb fell on my car. The 0.9 and the zero together constitute a belief function.

Fuzzy Logic

The concept of Fuzzy Logic (FL) was conceived by Lotfi Zadeh, a professor at the University of California at Berkley, and presented not as a control methodology, but as a way of processing data by allowing partial set membership rather than crisp set membership or non-membership. This approach to set theory was not applied to control systems until the 70's due to insufficient small-computer capability prior to that time. Professor Zadeh reasoned that people do not require precise, numerical information input, and yet they are capable of highly adaptive control. If feedback controllers could be programmed to accept noisy, imprecise input, they would be much

more effective and perhaps easier to implement. Unfortunately, U.S. manufacturers have not been so quick to embrace this technology while the Europeans and Japanese have been aggressively building real products around it.

What is Fuzzy logic?

In this context, FL is a problem-solving control system methodology that lends itself to implementation in systems ranging from simple, small, embedded micro-controllers to large, networked, multi-channel PC or workstation-based data acquisition and control systems. It can be implemented in hardware, software, or a combination of both. FL provides a simple way to arrive at a definite conclusion based upon vague, ambiguous, imprecise, noisy, or missing input information. FL's approach to control problems mimics how a person would make decisions, only much faster.

How is FL different form conventional control methods?

FL incorporates a simple, rule-based IF X AND Y THEN Z approach to a solving control problem rather than attempting to model a system mathematically. The FL model is empirically-based, relying on an operator's experience rather than their technical understanding of the system. For example, rather than dealing with temperature control in terms such as "SP =500F", "T <1000F", or "210C <TEMP <220C", terms like "IF (process is too cool) AND (process is getting colder) THEN (add heat to the process)" or "IF (process is too hot) AND (process is heating rapidly) THEN (cool the process quickly)" are used. These terms are imprecise and yet very descriptive of what must actually happen. Consider what you do in the shower if the temperature is too cold: you will make the water comfortable very quickly with little trouble. FL is capable of mimicking this type of behavior but at very high rate.

How does it work?

FL requires some numerical parameters in order to operate such as what is considered significant error and significant rate-of-change-of-error, but exact values of these numbers are usually not critical unless very responsive performance is required in which case empirical tuning would determine them. For example, a simple temperature control system could use a single temperature feedback sensor whose data is subtracted from the command signal to compute "error" and then time-differentiated to yield the error slope or rate-of-change-of-error, hereafter called "error-dot". Error might have units of degs F and a small error considered to be 2F while a large error is 5F. The "error-dot" might then have units of degs/min with a small error-dot being 5F/min and a large one being 15F/min. These values don't have to be symmetrical and can be "tweaked" once the system is operating in order to optimize performance. Generally, FL is so forgiving that the system will probably work the first time without any tweaking.

LEARNING

Tom Mitchell (1998) Well-posed Learning Problem:

"***A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience".***

- Designing algorithms that allow a computer to learn
- Deals with automatic recognition of patterns (e.g. automatic recognition of faces, postal codes on envelopes, speech recognition),
- Prediction (e.g. predicting the stock market),
- Data mining (e.g. finding good customers, fraudulent transactions)
- Autonomous acquisition and integration of knowledge.
- Continuously self-improve and thereby offer increased efficiency and effectiveness.
- Learning resembles human approach to learn task
- Internet-related problems (spam filtering, searching, sorting, network security).

Learning process is the basis of knowledge acquisition process. Knowledge acquisition is the expanding the capabilities of a system or improving its performance at some specified task. So we can say knowledge acquisition is the goal oriented creation and refinement of knowledge. The acquired knowledge may consist of various facts, rules, concepts, procedures, heuristics, formulas, relationships or any other useful information. The terms of increasing levels of abstraction, knowledge includes data, information and Meta knowledge. Meta knowledge includes the ability to evaluate the knowledge available, the additional knowledge required and the systematic implied by the present rules.

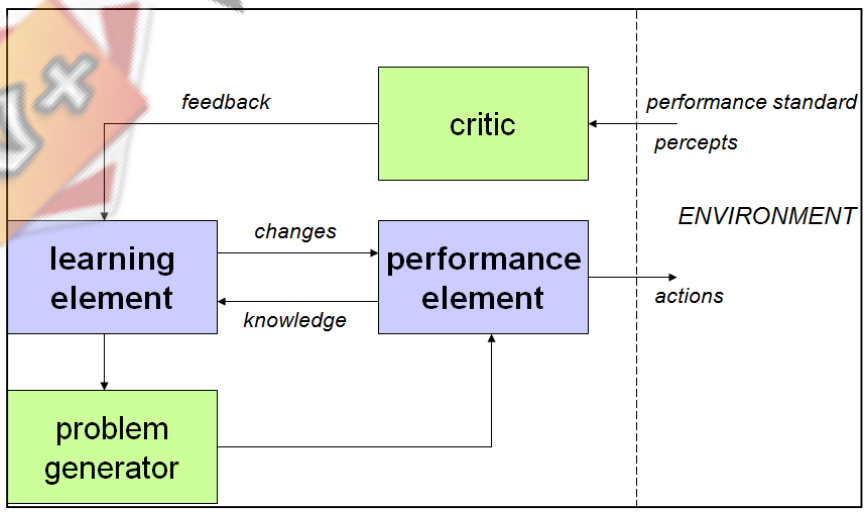


Figure Architecture of Machine Learning

Any system designed to create new knowledge and thereby improve its performance must include a set of data structures that represents the system's present level of expertise and a task algorithm that uses the rules to guide the system's problem solving activity.

Hence the inputs may be any types of inputs, those are executed for solution of a problem. Those inputs are processed to get the corresponding results. The learning element learns some sort of knowledge by the knowledge acquisition techniques. The acquired knowledge may be required for a same problem in future, for which that problem can be easily solved.

Every learning model must contain implicit or explicit restrictions on the class of functions that can learn. Among the set of all possible functions, we are particularly interested in a subset that contains all the tasks involved in intelligent behaviour. Examples of such tasks include visual perception, auditory perception, planning, control etc. The set does not just include specific visual perception tasks, but the set of all the tasks that an intelligent agent should be able to learn. Although we may like to think that the human brain is some what general purpose, it is extremely restricted in its ability to learn high dimensional functions.

Classification of Learning

The process of learning may be of various types. One can develop learning taxonomies based on the type of knowledge representation used (predicate calculus, rules, frames, scripts etc), the type of knowledge learned (game playing, problem solving) or by areas of application (medical diagnosis, engineering etc). Generally learning may be of two types like single agent learning and multi-agent learning.

A general architecture of learning process is given figure .

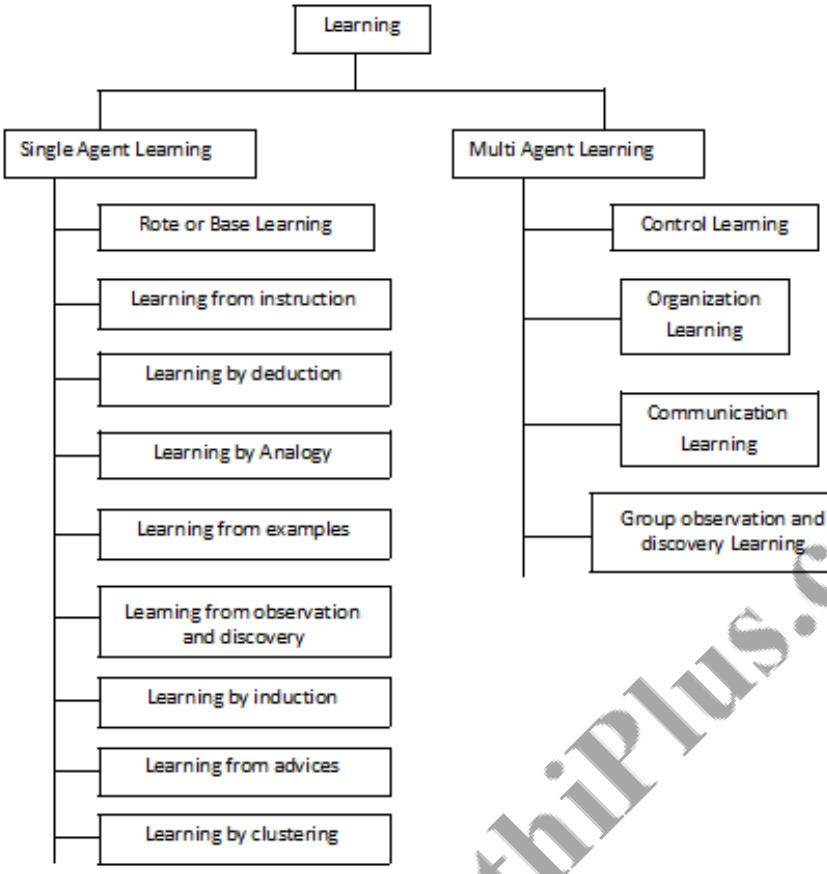


Figure: Learning Classification

Single Agent Learning

Over the last four decades, machine learning's primary interest has been single agent learning. Single agent learning involves improving the performance or increasing the knowledge of a single agent. An improvement in performance or an increase in knowledge allows the agent to solve past problems with better quality or efficiency. An increase in knowledge may also allow the agent to solve new problems. An increase in performance is not necessarily due to an increase in knowledge. It may be brought about simply by rearranging the existing knowledge or utilizing it in a different manner. Single agent learning systems may be classified according to their underlying learning strategies. These strategies are classified as follows.

Rote Learning

This strategy does not require the learning system to transform or infer knowledge. It 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. It includes learning my imitation, simple memorization and learning by being

performed. For example we may use this type of learning when we memorize multiplication tables. In this method we store the previous computed values, for which we do not have for recomputed them later. Also we can say rote learning is one type of existing or base learning. For example, in our childhood, we have the knowledge that "sun rises in the east". S o in our later stage of learning we can easily memorize the thing. Hence in this context, a system may simply memorize previous solutions and recall them when confronted with the same problem.

Learning from Instruction

This strategy also known as learning by being told or learning by direct instruction. It requires the learning system to select and transform knowledge into a usable form and then integrate it into the existing knowledge of the system. It is a more complex form of learning. This learning technique requires more inference than rote learning. It includes learning from teachers and learning by using books, publications and other types of instructions.

Learning by Deduction

This process is accomplished through a sequence of deductive inference steps using known facts. From the known facts, new facts or relationships are logically derived. Using this strategy, the learning system derives new facts from existing information or knowledge by employing deductive inference. It requires more inferences than other techniques. The inference method used is a deductive type, which is a valid form of inference. For example we can say x is the cousin of y if we have the knowledge of x's and y's parents and the rules for cousin relationships. The learner draws deductive inferences from the knowledge and reformulates them in the form of useful conclusions which preserve the information content of the original data.

Learning by Analogy

It is a process of learning a new concept or solution through the use of similar known concepts or solutions. We make frequent use of analogical learning. The first step is inductive inference, required to find a common substructure between the problem domain and one of the analogous domains stored in the learner's existing knowledge base. Example of learning by analogy may include the driving technique of vehicles. If we know the driving procedure of a bike, then when we will drive a car then some sort of previous learning procedures we may employ. Similarly for driving a bus or truck, we may use the procedure for driving a car.

Learning from Examples

In this process of learning it includes the learning through various interactive and innovative examples. This strategy, also called concept acquisition. It requires the learning system to induce general class or concept descriptions from examples. Since the learning system does not have prior or analogous knowledge of the concept area, the amount of inferencing is greater than both learning by deduction and analogy. For solving a newly designed problem we may use its corresponding old examples.

Learning from Observations and Discovery

Using this strategy, the learning system must either induce class descriptions from observing the environment or manipulate the environment to acquire class descriptions or concepts. This is an unsupervised learning technique. It requires the greatest amount of inferencing among all of the different forms of learning. From an existing knowledge base, some new forms of discovery of knowledge may formed. The learning discovery process is very important in the respect of constructing new knowledge base.

Learning by Induction

Inductive learning is the system that tries to induce a general rule based on observed instances. In other words, the system tries to infer an association between specific inputs and outputs. In general, the input of the program is a set of training instance where the output is a method of classifying subsequent instance. For example, the input of the program may be color of types of fruits where the output may be the types of fruits those are useful for protein. Induction method involves the learning by examples, experimentation, observation and discovery. The search spaces encountered in learning tend to be extremely large, even by the standards of search based problem solving.

Learning from Advices

In this process we can learn through taking advice from others. The idea of advice taking learning was proposed in early 1958 by McCarthy. In our daily life, this learning process is quite common. Right from our parents, relatives to our teachers, when we start our educational life, we take various advices from others. All most all the initial things and all type of knowledges we acquire through the advices of others. We know the computer programs are written by programmers. When a programmer writes a computer program he or she gives many instructions to computer to follow, the same way a teacher gives his/her advice to his students. The computer follows the instructions given by the programmer. Hence, a kind of learning takes place when computer runs a particular program by taking advice from the

creator of the program.

Learning by Clustering

This process is similar to the inductive learning. Clustering is a process of grouping or classifying objects on the basis of a close association or shared characteristics. The clustering process is essentially required in a learning process in which similarity patterns are found among a group of objects. The program must discover for itself the natural classes that exist for the objects, in addition to a method for classifying instances. AUTOCLASS (Cheeseman et al., 1988) is one program that accepts a number of training cases and hypothesizes a set of classes. For any given case, the program provides a set of probabilities that predict into which classes the case is likely to be fall.

Multi Agent Learning

Distributed artificial intelligence (DAI) systems solve problems using multiple, cooperative agents. In these systems, control and information are often distributed among the agents. This reduces the complexity of each agent and allows agents to work in parallel and increases problem solving speed. Also each agent has resource limitations which could limit the ability of a single agent system to solve large, complex problems. Allowing multiple agents to work on these types of problems may be the only way to realistically solve them. In general, multiple agent learning involves improving the performance of the group of agents as a whole or increasing the domain knowledge of the group. It also includes increasing communication knowledge. An increase in communication knowledge can lead to an increase in performance by allowing the agents to communicate in a more efficient manner. In the context of improving the performance of a group of agents, allowing individual agents to improve their performance may not be enough to improve the performance of the group. To apply learning to the overall group performance, the agents need to adapt and learn to work with the each other. The agents may not need to learn more about the domain, as in the traditional sense of machine learning, to improve group performance. In fact to improve the performance of the group, the agents may only need to learn to work together and not necessarily improve their individual performance. In addition, not all the agents must be able to learn or adapt to allow the group to improve.

Control Learning

Learning and adapting to work with other agents involves adjusting the control of each agent's problem solving plan. Different tasks may have to be solved in a specific sequence. If the tasks

are assigned to separate agents, the agents must work together to solve the tasks. Learning which agents are typically assigned different types of tasks will allow each agent to select other agents to work with on different tasks. Teams can be formed based on the type of task to be solved. Some of the issues involved are the type, immediacy and importance of task, as well as each agent's task solving ability, capability, reliability and past task assignments. Each team member's plan would be adjusted according to the other agent's plans.

Organization Learning

Learning what type of information and knowledge each agent possesses allows for an increase in performance by specifying the long term responsibilities of each agent. By assigning different agents different responsibilities, the group of agents can improve group performance by providing a global strategy. Organizing the responsibilities reduces the working complexity of each agent.

Communication Learning

Learning what type of information, knowledge reliability and capability each agent possesses allows for an increase in performance by allowing improved communication. Directly addressing the best agent for needed information or knowledge allows for more efficient communication among he agents.

Group Observation and Discovery Learning

Individual agents incorporate different information and knowledge. Combining this differing information and knowledge may assist in the process of learning new class descriptions or concepts that could not have been learned by the agents separately. This type of learning is more effective than the others. The observation towards the procedure will be focused by a group of agents. When a group of different visions will reach, at that point of view a newly interactive procedure will be found out; which is the discovery of all the agents.

Explanation based Learning

Explanation based learning has ability to learn from a single training instance. Instead of taking more examples the explanation based learning is emphasized to learn a single, specific example.

Machine Learning

Machine learning is a branch of science that deals with programming the systems in such a way that they automatically learn and improve with experience. Here, learning means recognizing and understanding the input data and making wise decisions based on the supplied data.

It is very difficult to cater to all the decisions based on all possible inputs. To tackle this problem, algorithms are developed. These algorithms build knowledge from specific data and past experience with the principles of statistics, probability theory, logic, combinatorial optimization, search, reinforcement learning, and control theory.

The developed algorithms form the basis of various applications such as:

- Vision processing
- Language processing
- Forecasting (e.g., stock market trends)
- Pattern recognition
- Games
- Data mining
- Expert systems
- Robotics

Machine learning is a vast area and it is quite beyond the scope of this tutorial to cover all its features. There are several ways to implement machine learning techniques, however the most commonly used ones are **supervised** and **unsupervised learning**.

Supervised Learning

Supervised learning deals with learning a function from available training data. A supervised learning algorithm analyzes the training data and produces an inferred function, which can be used for mapping new examples. Common examples of supervised learning include:

- Classifying e-mails as spam,
- Labeling web pages based on their content, and
- Voice recognition.

There are many supervised learning algorithms such as neural networks, Support Vector Machines (SVMs), and Naive Bayes classifiers. Mahout implements Naive Bayes classifier.

Unsupervised Learning

Unsupervised learning makes sense of unlabeled data without having any predefined dataset for its training. Unsupervised learning is an extremely powerful tool for analyzing available data and look for patterns and trends. It is most commonly used for clustering similar input into logical groups. Common approaches to unsupervised learning include:

- k-means
- self-organizing maps, and
- hierarchical clustering

Recommendation

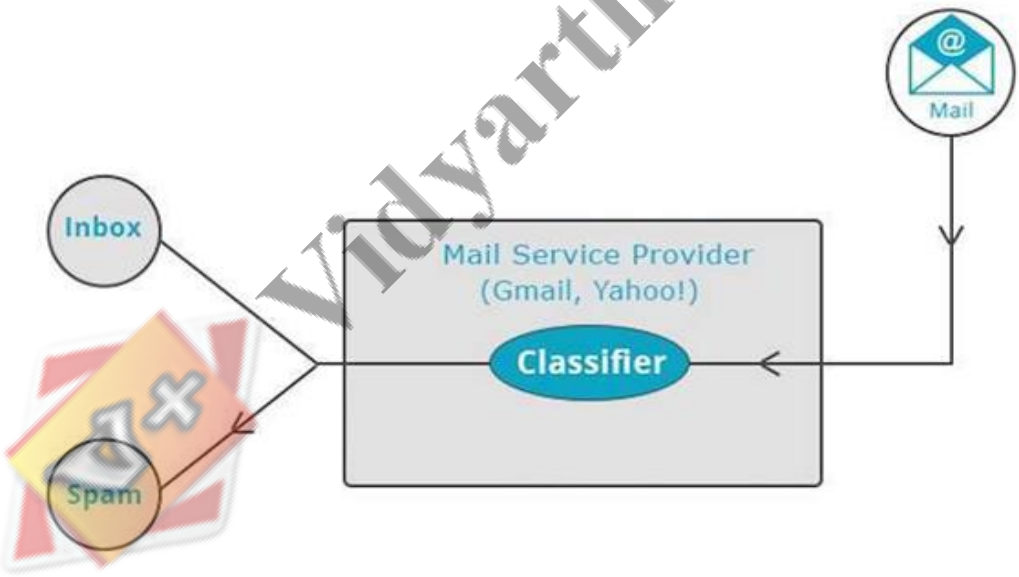
Recommendation is a popular technique that provides close recommendations based on user information such as previous purchases, clicks, and ratings.

- Amazon uses this technique to display a list of recommended items that you might be interested in, drawing information from your past actions. There are recommender engines that work behind Amazon to capture user behavior and recommend selected items based on your earlier actions.
- Facebook uses the recommender technique to identify and recommend the "people you may know list".

Classification

Classification, also known as **categorization**, is a machine learning technique that uses known data to determine how the new data should be classified into a set of existing categories. Classification is a form of supervised learning.

- Mail service providers such as Yahoo! and Gmail use this technique to decide whether a new mail should be classified as a spam. The categorization algorithm trains itself by analyzing user habits of marking certain mails as spams. Based on that, the classifier decides whether a future mail should be deposited in your inbox or in the spams folder.

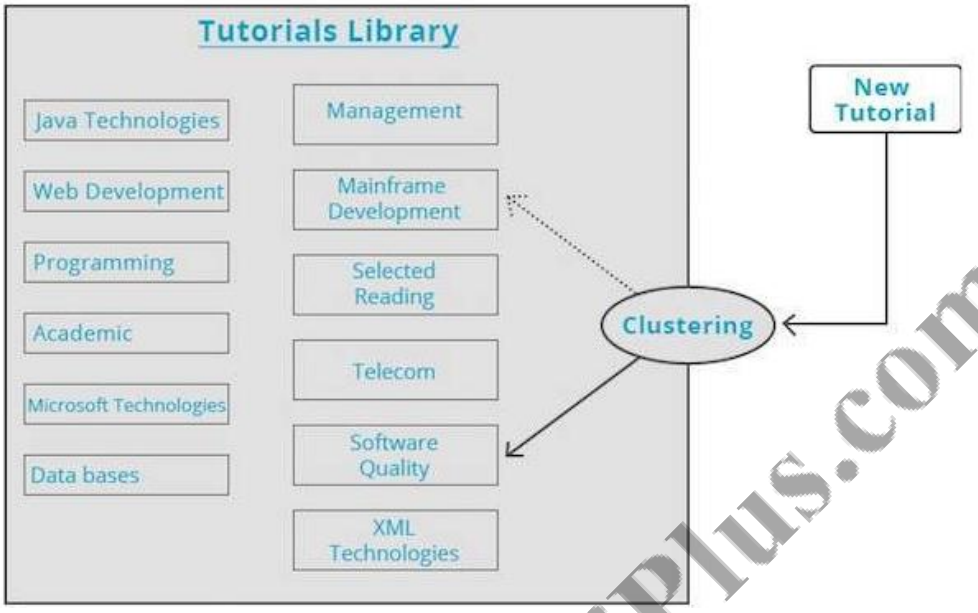


Clustering

Clustering is used to form groups or clusters of similar data based on common characteristics. Clustering is a form of unsupervised learning.

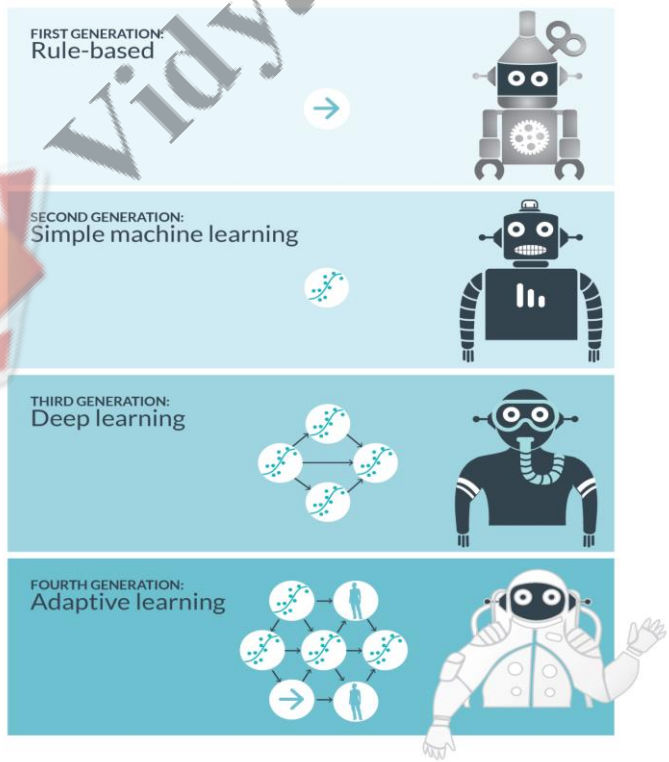
- Search engines such as Google and Yahoo! use clustering techniques to group data with similar characteristics.
- Newsgroups use clustering techniques to group various articles based on related topics.

The clustering engine goes through the input data completely and based on the characteristics of the data, it will decide under which cluster it should be grouped. Take a look at the following example.



Learning Generations

Four generations of machine intelligence



1st generation: Rules

The first generation of machine intelligence meant that people manually created rules. For example, in text analytics someone might create a rule that the word "Ford" followed by "Focus" meant that "Ford" referred to a car, and they would create a separate rule that "Ford" preceded by "Harrison" meant that "Ford" referred to a person.

2nd generation: Simple machine learning

The dominant form of machine intelligence today is simple machine learning. Simple machine learning uses statistical methods to make decisions about data processing. For example, a sentence might have the word "Ford" labeled as a car, and the machine learning algorithm will learn by itself that the following word "Focus" is evidence that "Ford" is a car in this context. Simple machine learning can be fast, provided that you already have labeled examples for 'supervised learning'. It also tends to be more accurate, because statistics are usually better than human intuition in deciding which features (like words and phrases) matter.

3rd generation: Deep learning

Deep learning (deep structured learning, hierarchical learning or deep machine learning) is a branch of machine learning based on a set of algorithms that attempt to model high-level abstractions in data by using multiple processing layers, with complex structures.

4th generation: Adaptive learning

Adaptive learning is an educational method which uses computers as interactive teaching devices, and to orchestrate the allocation of human and mediated resources according to the unique needs of each learner. The technology encompasses aspects derived from various fields of study including computer science, education, psychology, and brain science.

Adaptive learning has been partially driven by a realization that tailored learning cannot be achieved on large-scale using traditional, non-adaptive approaches. Adaptive learning systems endeavor to transform the learner from passive receptor of information to collaborator in the educational process. Adaptive learning systems' primary application is in education, but another popular application is business training. They have been designed as desktop computer applications, web applications, and are now being introduced into overall curricula. Adaptive learning has been implemented in several kinds of educational systems such as adaptive hypermedia, intelligent tutoring systems, computerized adaptive testing, and computer-based pedagogical agents.

Expert Systems

Definition

It's always good to start with a definition. Here are two definitions of an expert system:

> *"A model and associated procedure that exhibits, within a specific domain, a degree of expertise in problem solving that is comparable to that of a human expert". Ignizio*

> *"An expert system is a computer system which emulates the decision-making ability of a human expert". Giarratono*

Simply put, an expert system contains knowledge derived from an expert in some narrow domain. This knowledge is used to help individuals using the expert system to solve some problem. I mention narrow domain since it is quite difficult to encode enough knowledge into a system so that it may solve a variety of problems. We have not reached the point yet where this can be done.

AI programs that achieve expert-level competence in solving problems in task areas by bringing to bear a body of knowledge about specific tasks are called *knowledge-based* or *expert systems*. Often, the term expert systems is reserved for programs whose knowledge base contains the knowledge used by human experts, in contrast to knowledge gathered from textbooks or non-experts. More often than not, the two terms, expert systems (ES) and knowledge-based systems (KBS), are used synonymously. Taken together, they represent the most widespread type of AI application. The area of human intellectual endeavor to be captured in an expert system is called the *task domain*. *Task* refers to some goal-oriented, problem-solving activity. *Domain* refers to the area within which the task is being performed. Typical tasks are diagnosis, planning, scheduling, configuration and design.

Building an expert system is known as *knowledge engineering* and its practitioners are called *knowledge engineers*. The knowledge engineer must make sure that the computer has all the knowledge needed to solve a problem. The knowledge engineer must choose one or more forms in which to represent the required knowledge as symbol patterns in the memory of the computer -- that is, he (or she) must choose a *knowledge representation*. He must also ensure that the computer can use the knowledge efficiently by selecting from a handful of *reasoning methods*. The practice of knowledge engineering is described later. We first describe the components of expert systems.

Components of an Expert System

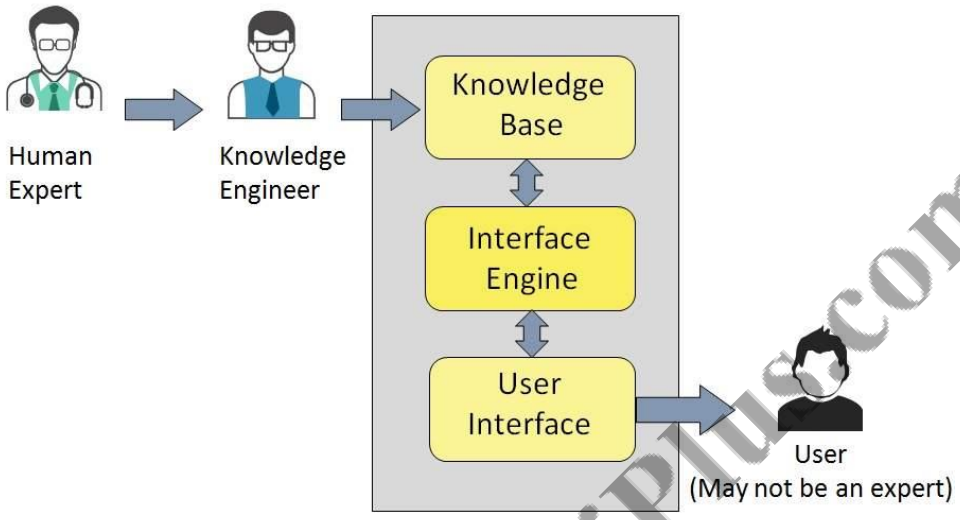
An expert system consists of these three components:

- Structure
- Knowledge base

- Inference engine

Structure

Expert systems generally follow this structure:



Each component will be described in the following sections.

The ***knowledge base*** of expert systems contains both factual and heuristic knowledge. *Factual knowledge* is that knowledge of the task domain that is widely shared, typically found in textbooks or journals, and commonly agreed upon by those knowledgeable in the particular field.

Heuristic knowledge is the less rigorous, more experiential, more judgmental knowledge of performance. In contrast to factual knowledge, heuristic knowledge is rarely discussed, and is largely individualistic. It is the knowledge of good practice, good judgment, and plausible reasoning in the field. It is the knowledge that underlies the "art of good guessing."

Knowledge representation formalizes and organizes the knowledge. One widely used representation is the *production rule*, or simply *rule*. A rule consists of an IF part and a THEN part (also called a *condition* and an *action*). The IF part lists a set of conditions in some logical combination. The piece of knowledge represented by the production rule is relevant to the line of reasoning being developed if the IF part of the rule is satisfied; consequently, the THEN part can be concluded, or its problem-solving action taken. Expert systems whose knowledge is represented in rule form are called *rule-based systems*.

Another widely used representation, called the *unit* (also known as *frame*, *schema*, or *list structure*) is based upon a more passive view of knowledge. The unit is an assemblage of associated symbolic knowledge about an entity to be represented. Typically, a unit consists of a list of properties of the entity and associated values for those properties.

Since every task domain consists of many entities that stand in various relations, the properties can also be used to specify relations, and the values of these properties are the names of other

units that are linked according to the relations. One unit can also represent knowledge that is a "special case" of another unit, or some units can be "parts of" another unit.

The *problem-solving model*, or *paradigm*, organizes and controls the steps taken to solve the problem. One common but powerful paradigm involves chaining of IF-THEN rules to form a line of reasoning. If the chaining starts from a set of conditions and moves toward some conclusion, the method is called *forward chaining*. If the conclusion is known (for example, a goal to be achieved) but the path to that conclusion is not known, then reasoning backwards is called for, and the method is *backward chaining*. These problem-solving methods are built into program modules called *inference engines* or *inference procedures* that manipulate and use knowledge in the knowledge base to form a line of reasoning.

The *knowledge base* an expert uses is what he learned at school, from colleagues, and from years of experience. Presumably the more experience he has, the larger his store of knowledge. Knowledge allows him to interpret the information in his databases to advantage in diagnosis, design, and analysis.

Though an expert system consists primarily of a knowledge base and an inference engine, a couple of other features are worth mentioning: reasoning with uncertainty, and explanation of the line of reasoning.

Knowledge is almost always incomplete and uncertain. To deal with uncertain knowledge, a rule may have associated with it a *confidence factor* or a weight. The set of methods for using uncertain knowledge in combination with uncertain data in the reasoning process is called *reasoning with uncertainty*. An important subclass of methods for reasoning with uncertainty is called "fuzzy logic," and the systems that use them are known as "fuzzy systems."

The most important ingredient in any expert system is knowledge. The power of expert systems resides in the specific, high-quality knowledge they contain about task domains. AI researchers will continue to explore and add to the current repertoire of knowledge representation and reasoning methods. But in knowledge resides the power. Because of the importance of knowledge in expert systems and because the current knowledge acquisition method is slow and tedious, much of the future of expert systems depends on breaking the knowledge acquisition bottleneck and in codifying and representing a large knowledge infrastructure.

Knowledge engineering

is the art of designing and building expert systems, and knowledge engineers are its practitioners. Gerald M. Weinberg said of programming in *The Psychology of Programming*: "'Programming,' -- like 'loving,' -- is a single word that encompasses an infinitude of activities" (Weinberg 1971). Knowledge engineering is the same, perhaps more so. We stated earlier that knowledge engineering is an applied part of the science of artificial intelligence which, in turn, is a part of computer science. Theoretically, then, a knowledge engineer is a computer scientist who knows how to design and implement programs that incorporate artificial intelligence techniques. The nature of knowledge engineering is changing, however, and a new breed of knowledge engineers is emerging. We'll discuss the evolving nature of knowledge engineering later.

Today there are two ways to build an expert system. They can be built from scratch, or built using a piece of development software known as a "tool" or a "shell." Before we discuss these tools, let's briefly discuss what knowledge engineers do. Though different styles and methods of knowledge engineering exist, the basic approach is the same: a knowledge engineer interviews and observes a human expert or a group of experts and learns what the experts know, and how they reason with their knowledge. The engineer then translates the knowledge into a computer-usable language, and designs an inference engine, a reasoning structure, that uses the knowledge appropriately. He also determines how to integrate the use of uncertain knowledge in the reasoning process, and what kinds of explanation would be useful to the end user.

Next, the inference engine and facilities for representing knowledge and for explaining are programmed, and the domain knowledge is entered into the program piece by piece. It may be that the inference engine is not just right; the form of knowledge representation is awkward for the kind of knowledge needed for the task; and the expert might decide the pieces of knowledge are wrong. All these are discovered and modified as the expert system gradually gains competence.

The discovery and accumulation of techniques of machine reasoning and knowledge representation is generally the work of artificial intelligence research. The discovery and accumulation of knowledge of a task domain is the province of domain experts. Domain knowledge consists of both formal, textbook knowledge, and experiential knowledge -- the *expertise* of the experts.

Tools, Shells, and Skeletons

Compared to the wide variation in domain knowledge, only a small number of AI methods are known that are useful in expert systems. That is, currently there are only a handful of ways in which to represent knowledge, or to make inferences, or to generate explanations. Thus, systems can be built that contain these useful methods without any domain-specific knowledge. Such systems are known as *skeletal systems, shells*, or simply *AI tools*.

Building expert systems by using shells offers significant advantages. A system can be built to perform a unique task by entering into a shell all the necessary knowledge about a task domain. The inference engine that applies the knowledge to the task at hand is built into the shell. If the program is not very complicated and if an expert has had some training in the use of a shell, the expert can enter the knowledge himself.

Many commercial shells are available today, ranging in size from shells on PCs, to shells on workstations, to shells on large mainframe computers. They range in price from hundreds to tens of thousands of dollars, and range in complexity from simple, forward-chained, rule-based systems requiring two days of training to those so complex that only highly trained knowledge engineers can use them to advantage. They range from general-purpose shells to shells custom-tailored to a class of tasks, such as financial planning or real-time process control.

Although shells simplify programming, in general they don't help with knowledge acquisition. *Knowledge acquisition* refers to the task of endowing expert systems with knowledge, a task

currently performed by knowledge engineers. The choice of reasoning method, or a shell, is important, but it isn't as important as the accumulation of high-quality knowledge. The power of an expert system lies in its store of knowledge about the task domain -- the more knowledge a system is given, the more competent it becomes.

Inference Engine

The inference engine controls overall execution of the rules. It searches through the knowledge base, attempting to pattern match facts or knowledge present in memory to the antecedents of rules. If a rule's antecedent is satisfied, the rule is ready to fire and is placed in the agenda. When a rule is ready to fire it means that since the antecedent is satisfied, the consequent can be executed.

Salience is a mechanism used by some expert systems to add a procedural aspect to rule inferencing. Certain rules may be given a higher salience than others, which means that when the inference engine is searching for rules to fire, it places those with a higher salience at the top of the agenda.

Use of efficient procedures and rules by the Interface Engine is essential in deducting a correct, flawless solution.

In case of knowledge-based ES, the Interface Engine acquires and manipulates the knowledge from the knowledge base to arrive at a particular solution.

In case of rule based ES, it −

- Applies rules repeatedly to the facts, which are obtained from earlier rule application.
- Adds new knowledge into the knowledge base if required.
- Resolves rules conflict when multiple rules are applicable to a particular case.

To recommend a solution, the interface engine uses the following strategies −

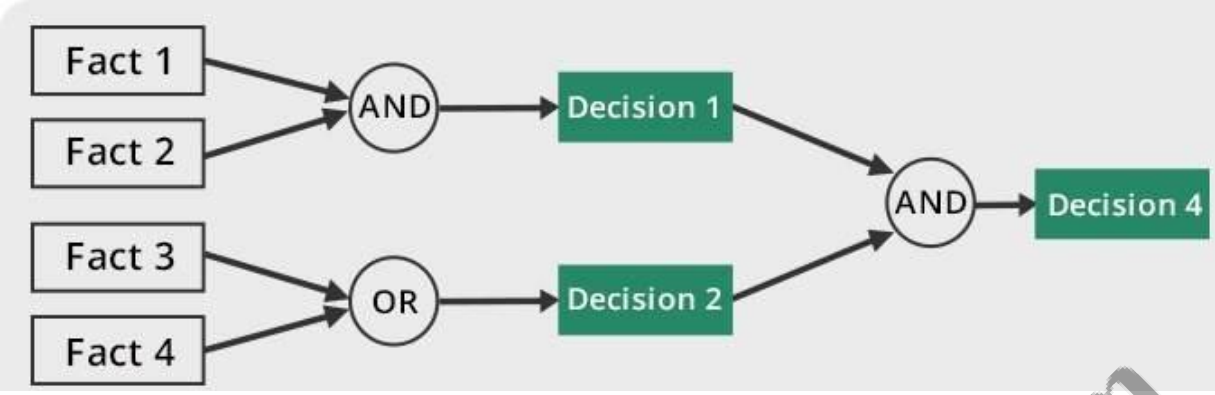
- Forward Chaining
- Backward Chaining

Forward Chaining

It is a strategy of an expert system to answer the question, **“What can happen next?”**

Here, the interface engine follows the chain of conditions and derivations and finally deduces the outcome. It considers all the facts and rules, and sorts them before concluding to a solution.

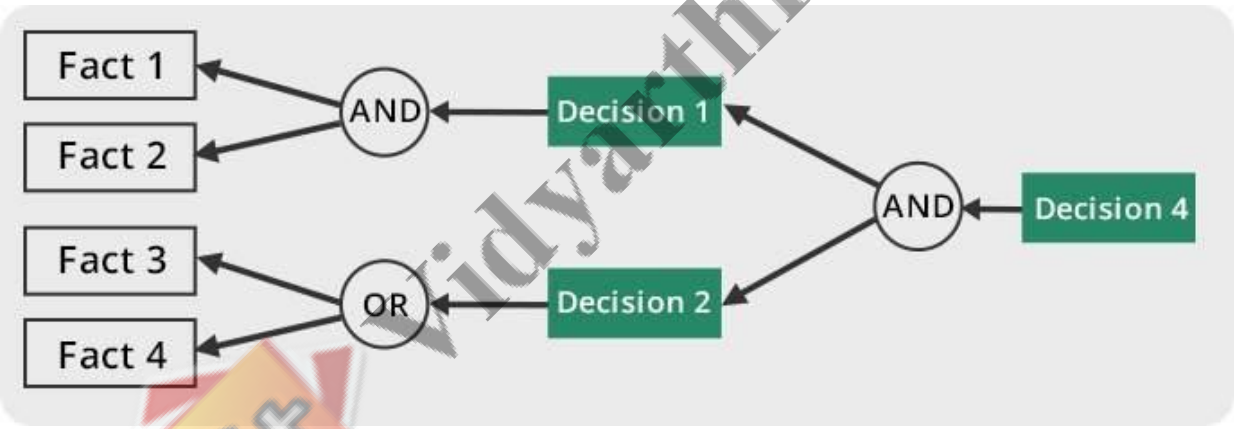
This strategy is followed for working on conclusion, result, or effect. For example, prediction of share market status as an effect of changes in interest rates.



Backward Chaining

With this strategy, an expert system finds out the answer to the question, **"Why this happened?"**

On the basis of what has already happened, the interface engine tries to find out which conditions could have happened in the past for this result. This strategy is followed for finding out cause or reason. For example, diagnosis of blood cancer in humans.



Inferencing is to computers what reasoning is to humans.

Uncertainty

Uncertainty in expert systems can be handled in a variety of approaches. Here are some of them, with brief descriptions:

- Certainty factors
- Dempster-Shafer theory
- Bayesian network
- Fuzzy logic

Certainty Factors are used as a degree of confirmation of a piece of evidence. Mathematically, a certainty factor is the measure of belief minus the measure of disbelief. Here is an example:

```
If the light is green
then
OK to cross the street cf 0.9
```

The rule in the example says: I am 90% certain that it is safe to cross the street when the light is green.

There are certain advantages and disadvantages to certainty factors. They are easy to compute and can be used to easily propagate uncertainty through the system. However, they are created partly ad hoc. Also, the certainty factor of two rules in an inference chain is calculated as independent probabilities.

Dempster-Shafer Theory does not force belief to be assigned to ignorance or refutation of a hypothesis. For example, belief of 0.7 in falling asleep in class does not mean that the chance of not falling asleep in class is 0.3

Bayesian Networks are based on Bayes Theorem:

$$P(H|E) = \frac{P(E|H)P(H)}{P(E)}$$

Bayes Theorem gives the probability of event H given that event E has occurred. Bayesian networks have their use, but are often not practical for large systems. There is also a problem with the uncertainty of user responses.

Fuzzy Logic

Fuzzy logic is a superset of conventional (Boolean) logic that has been extended to handle the concept of a partial truth -- truth values between completely true and completely false. In fuzzy logic, everything is a matter of degree.

Some people think that fuzzy logic is a contradiction of terms. Fuzzy logic is a logic **OF** fuzziness, not a logic which is **ITSELF** fuzzy. Fuzzy sets and logic are used to represent uncertainty, which is crucial for the management of real systems. A fuzzy expert system is an expert system that uses a collection of fuzzy membership functions and rules to reason about data. Every rules fires to some degree.

The fuzzy inferencing process becomes much more complicated, expanding to 4 steps:

1. Fuzzification
2. Inference
3. Composition
4. Defuzzification

Building an Expert System

There are basically 4 steps to building an expert system.

1. Analysis
2. Specification
3. Development
4. Deployment

The spiral model is normally used to implement this approach. The spiral model of developing software is fairly common these days. Expert system development can be modeled as a spiral, where each circuit adds more capabilities to the system. There are other approaches, such as the incremental or linear model, but we prefer the spiral model.

Analysis

The purpose of analysis is to identify a potential application. Possible applications include diagnostics, a controller, etc. During analysis the developer must also assess the suitability of knowledge-engineering technology for this application. You must ask yourself the question *Will something else work better?* This is true for applying any type of artificial intelligence to solve a problem. If there is a numerical method or heuristic that is well established, than stick with that approach and use artificial intelligence to solve problems which are difficult.

Specification

The specification step is where the developer defines what the expert system will do. Here the developer must also work with the expert to learn enough about the task to plan system development. The expert is a human who is identified as being the domain expert in a particular field. The developer must familiarize himself with the problem so that system development can be performed. The developer will spend a significant amount of time in this phase acquiring knowledge.

Defining what an expert system should do can be challenging. It may be difficult to obtain reliable information. Some experts may solve problems differently, or tell the developer what they think he wants to hear. The experts may envision a different functionality for the system than the developer, who better understands the limitations of the software. It is also important to assure the experts that the purpose of the expert system is not to replace the experts, but to proliferate their knowledge and expertise throughout the organization. Also, once an expert system is developed, it cannot create new ways to solve problems. It is up to the human experts to continually refine their knowledge and find better ways of solving problems.

Development

The development step consists of several important tasks. Here, the developer must learn how the expert performs the task (knowledge acquisition) in a variety of cases. There are basically

three kinds of cases the developer should discuss with the expert: current, historical, and hypothetical. Current cases can be covered by watching the expert perform a task. Historical cases can be discussed by discussing with the expert a task that was performed in the past. And, hypothetical cases can be covered by having the expert describe how a task should be performed in a hypothetical situation.

The knowledge acquisition process, which started in the specification phase, continues into the development phase. The developer must extract knowledge from the previous case discussions. The types of knowledge the developer looks for can be grouped into three categories: strategic, judgmental, and factual. Strategic knowledge is used to help create a flow chart of the system. Judgemental knowledge usually helps define the inference process and describes the reasoning process used by the expert. Finally, factual knowledge describes the characteristics and important attributes of objects in the system.

Next, a conceptual model of the expert system must be developed. This conceptual model is a framework which consists of high-level descriptions of the tasks and situations. In this framework, the developer must:

- Decide how the inference, representation, and control structure can be used to implement.
- Build the knowledge base.
- Verify and validate (am I building the product right? Am I building the right product?).

Deployment

In the deployment phase the developer installs the system for routine use. He also fixes bugs, updates, and enhances the expert system.

Some Notes on Verification & Validation

There are two areas the expert system developer should take note of in the verification & validation process which are peculiar to expert systems: inconsistencies and incompleteness.

Inconsistencies can be caused by redundant rules, conflicting rules, subsumed rules, unnecessary premise clauses, and circular rule chains. Incompleteness can be caused by unachievable antecedents or consequences and unreferenced or illegal values.

Expert Systems Limitations

No technology can offer easy and complete solution. Large systems are costly, require significant development time, and computer resources. ESs have their limitations which include –

- Limitations of the technology
- Difficult knowledge acquisition
- ES are difficult to maintain
- High development costs

Applications of Expert System

The spectrum of applications of expert systems technology to industrial and commercial problems is so wide as to defy easy characterization. The applications find their way into most areas of knowledge work. They are as varied as helping salespersons sell modular factory-built homes to helping NASA plan the maintenance of a space shuttle in preparation for its next flight.

Applications tend to cluster into seven major classes.

Diagnosis and Troubleshooting of Devices and Systems of All Kinds

This class comprises systems that deduce faults and suggest corrective actions for a malfunctioning device or process. Medical diagnosis was one of the first knowledge areas to which ES technology was applied (for example, see Shortliffe 1976), but diagnosis of engineered systems quickly surpassed medical diagnosis. There are probably more diagnostic applications of ES than any other type. The diagnostic problem can be stated in the abstract as: given the evidence presenting itself, what is the underlying problem/reason/cause?

Planning and Scheduling

Systems that fall into this class analyze a set of one or more potentially complex and interacting goals in order to determine a set of actions to achieve those goals, and/or provide a detailed temporal ordering of those actions, taking into account personnel, materiel, and other constraints. This class has great commercial potential, which has been recognized. Examples involve airline scheduling of flights, personnel, and gates; manufacturing job-shop scheduling; and manufacturing process planning.

Configuration of Manufactured Objects from Subassemblies

Configuration, whereby a solution to a problem is synthesized from a given set of elements related by a set of constraints, is historically one of the most important of expert system applications. Configuration applications were pioneered by computer companies as a means of facilitating the manufacture of semi-custom minicomputers (McDermott 1981). The technique has found its way into use in many different industries, for example, modular home building, manufacturing, and other problems involving complex engineering design and manufacturing.

Financial Decision Making

The financial services industry has been a vigorous user of expert system techniques. Advisory programs have been created to assist bankers in determining whether to make loans to businesses and individuals. Insurance companies have used expert systems to assess the risk presented by the customer and to determine a price for the insurance. A typical application in the financial markets is in foreign exchange trading.

Knowledge Publishing

This is a relatively new, but also potentially explosive area. The primary function of the expert system is to deliver knowledge that is relevant to the user's problem, in the context of the user's problem. The two most widely distributed expert systems in the world are in this category. The first is an advisor which counsels a user on appropriate grammatical usage in a text. The second is a tax advisor that accompanies a tax preparation program and advises the user on tax strategy, tactics, and individual tax policy.

Process Monitoring and Control

Systems falling in this class analyze real-time data from physical devices with the goal of noticing anomalies, predicting trends, and controlling for both optimality and failure correction. Examples of real-time systems that actively monitor processes can be found in the steel making and oil refining industries.

Design and Manufacturing

These systems assist in the design of physical devices and processes, ranging from high-level conceptual design of abstract entities all the way to factory floor configuration of manufacturing processes.

Expert System Technology

There are several levels of ES technologies available. Expert systems technologies include −

- **Expert System Development Environment** − The ES development environment includes hardware and tools. They are −
    - Workstations, minicomputers, mainframes.
    - High level Symbolic Programming Languages such as **LIS**t **P**rogramming (LISP) and **PRO**grammation en **LOG**ique (PROLOG).
    - Large databases.
- **Tools** − They reduce the effort and cost involved in developing an expert system to large extent.
    - Powerful editors and debugging tools with multi-windows.
    - They provide rapid prototyping
    - Have Inbuilt definitions of model, knowledge representation, and inference design.
- **Shells** − A shell is nothing but an expert system without knowledge base. A shell provides the developers with knowledge acquisition, inference engine, user interface, and explanation facility. For example, few shells are given below −
    - Java Expert System Shell (JESS) that provides fully developed Java API for creating an expert system.
    - *Vidwan*, a shell developed at the National Centre for Software Technology, Mumbai in 1993. It enables knowledge encoding in the form of IF-THEN rules.

Sample Expert Systems - Dendral, Mycin, XCON

DENDRAL

DENDRAL is a program that analyses organic compounds to determine their structure. It is one of the early example of a successful AI program . it uses a strategy called plan-generate-test in which a planning process that used constraint-satisfaction techniques, creates lists of recommended and contraindicated substructures.

MYCIN

Mycin is a program that diagnoses infectious diseases. It reasons backward from its goal of determining the cause of a patient illness. It attempts to solve its goal of recommending a therapy for a particular patient by first finding the cause of the patient's illness. It uses its production rule4s to reason backward from goals to clinical observations. To solve the top-level diagnostic goal, it looks for rules whose right sides suggest diseases. It then uses the left sides of those rules (the preconditions) to set up sub goals whose success would enable the rules to be invoked . these sub goals are again matched against rules, and their preconditions are used to set up additional sub goals.

Mycin is a well known rule based deduction system. Its expertise lies in the domain of bacterial Infections. Physicians usually must begin antibiotic treatment for patient who have bacterial infections without knowledge exactly which organism is the culprit. There is no time to wait for definitive laboratory culture evidence, which accumulates too slowly. For the desperately sick, therapy must begin at once – not 2 days from can either prescribe a broad – spectrum drug that covers all possibilities , or she can prescribed a better, disease – specific drug.

Mycin helps the physician to prescribe disease – specific drugs. Mycin in-forms it self about particular cases by requesting information from the physician about a patient's symptoms, general condition. History, and laboratory – test results that can be obtained easily and quickly. At each point, the question mycin asks is determined by Mycin's current hypothesis and the answers to all previous questions. Thus, the questions start as though taken from a checklist, but the questions then vary as evidence builds. Here is an except from a sample sessions:

What is the patient's Name ?

john Doe.

- Male or female?

Mael

- Age ?

He is 55.

- Have you obtained positive cultures indicating general type ?

yes

- What type of inflection is it ?

Primary bacteremia

- When did symptoms first appear ?

May 5

- Let's call the most positive culture CI. From what site was CI taken ?

from the blood.

- When ?

May 9

- Let's call the first significant organism from this culture UI. Do you know identify of UI?

no

- Is UI a rod or a coccus or something else ?

Rod

- What is the gram stain of UI?

Gram negative.

- Have there been positive culture from sites that are not associated with the

inflections about which you are seeking advice?

No

As we proceed through the processing stages of compute vision. We will no doubt be impressed by the similarities and parallel one can draw between vision processing and natural language processing . The - sensor stage in vision corresponds to speech recognization language understanding, the low and intermediate processing levels of vision correspond to syntactic and semantic language processing respectively, and high level processing, in both cases corresponds to the process of building and interpreting high level knowledge structures.

XCON

The **R1** (internally called **XCON**, for e**X**pert **CON**figurer) program was a production-rule-based system written in OPS5 by John P. McDermott of CMU in 1978 to assist in the ordering of DEC's VAX computer systems by automatically selecting the computer system components based on the customer's requirements.

In developing the system McDermott made use of experts from both DEC's PDP/11 and VAX computer systems groups. These experts sometimes even disagreed amongst themselves as to an optimal configuration. The resultant "sorting it out" had an additional benefit in terms of the quality of VAX systems delivered.

XCON first went into use in 1980 in DEC's plant in Salem, New Hampshire. It eventually had about 2500 rules. By 1986, it had processed 80,000 orders, and achieved 95-98% accuracy. It was estimated to be saving DEC $25M a year by reducing the need to give customers free components when technicians made errors, by speeding the assembly process, and by increasing customer satisfaction.

Before XCON, when ordering a VAX from DEC, every cable, connection, and bit of software had to be ordered separately. (Computers and peripherals were not sold complete in boxes as they are today). The sales people were not always very technically expert, so customers would find that they had hardware without the correct cables, printers without the correct drivers, a processor without the correct language chip, and so on. This meant delays and caused a lot of customer dissatisfaction and resultant legal action. XCON interacted with the sales person, asking critical questions before printing out a coherent and workable system specification/order slip.

XCON's success led DEC to rewrite XCON as **XSEL**- a version of XCON intended for use by DEC's sales force to aid a customer in properly configuring their VAX (so they wouldn't, say, choose a computer too large to fit through their doorway or choose too few cabinets for the components to fit in). Location problems and configuration were handled by yet another expert system, **XSITE**.

McDermott's 1980 paper on R1 won the AAAI Classic Paper Award in 1999. Legendarily, the name of R1 comes from McDermott, who supposedly said as he was writing it, "Three years ago I couldn't spell knowledge engineer, now I are one."