



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
**Institute of Management & Technology**  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

**GURU GOBIND SINGH  
INDRAPRASTHA UNIVERSITY**

**Paper Code: BCA 312  
Paper: Artificial Intelligence**

**UNIT - I**

Overview of A.I: Introduction to AI, Importance of AI, AI and its related field, AI techniques, Criteria for success.

Problems, problem space and search: Defining the problem as a state space search, Production system and its characteristics, Issues in the design of the search problem.

Heuristic search techniques: Generate and test, hill climbing, best first search technique, problem reduction, constraint satisfaction.

**UNIT - II**

Knowledge representation: Definition and importance of knowledge, Knowledge representation, various approaches used in knowledge representation, Issues in knowledge representation.

Using Predicate Logic: Representing Simple Facts in logic, Representing instances and is-a relationship, Computable function and predicate.

**UNIT - III**

Natural language processing: Introduction syntactic processing, Semantic processing, Discourse and pragmatic processing.

Learning: Introduction learning, Rote learning, Learning by taking advice, learning in problem solving, Learning from example-induction, Explanation based learning.

**UNIT - IV**

Expert System: Introduction, Representing using domain specific knowledge, Expert system shells. LISP and other AI Programming Language



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

## UNIT - I

### Overview of A.I: Introduction to AI

Artificial intelligence (AI) is the intelligence of machines and the branch of computer science that aims to create it. AI textbooks define the field as "the study and design of intelligent agents" where an intelligent agent is a system that perceives its environment and takes actions that maximize its chances of success. John McCarthy, who coined the term in 1956, defines it as "the science and engineering of making intelligent machines."

The field was founded on the claim that a central property of humans, intelligence—the sapience of *Homo sapiens*—can be so precisely described that it can be simulated by a machine.<sup>[5]</sup> This raises philosophical issues about the nature of the mind and the ethics of creating artificial beings, issues which have been addressed by myth, fiction and philosophy since antiquity. Artificial intelligence has been the subject of optimism, but has also suffered setbacks and, today, has become an essential part of the technology industry, providing the heavy lifting for many of the most difficult problems in computer science.

AI research is highly technical and specialized, *deeply* divided into subfields that often fail in the task of communicating with each other. Subfields have grown up around particular institutions, the work of individual researchers, and the solution of specific problems, resulting in longstanding differences of opinion about how AI should be done and the application of widely differing tools. The central problems of AI include such traits as reasoning, knowledge, planning, learning, communication, perception and the ability to move and manipulate objects. General

In the early 1980s, AI research was revived by the commercial success of expert systems,<sup>[1]</sup> a form of AI program that simulated the knowledge and analytical skills of one or more human experts. By 1985 the market for AI had reached over a billion dollars. At the same time, Japan's fifth generation computer project inspired the U.S and British governments to restore funding for academic research in the field. However, beginning with the collapse of the Lisp Machine market in 1987, AI once again fell into disrepute, and a second, longer lasting AI winter began.

In the 1990s and early 21st century, AI achieved its greatest successes, albeit somewhat behind the scenes. Artificial intelligence is used for logistics, data mining, medical diagnosis and many other areas throughout the technology industry.<sup>[9]</sup> The success was due to several factors: the increasing computational power of computers (see Moore's law), a greater emphasis on solving specific subproblems, the creation of new ties between AI and other fields working on similar problems, and a new commitment by researchers to solid mathematical methods and rigorous.

### Problems

"Can a machine act intelligently?" is still an open problem. Taking "A machine can act intelligently" as a working hypothesis, many researchers have attempted to build such a machine.

The general problem of simulating (or creating) intelligence has been broken down into a number of specific sub-problems. These consist of particular traits or capabilities that researchers would like an intelligent system to display. The traits described below have received the most attention.

### Deduction, reasoning, problem solving:-

Early AI researchers developed algorithms that imitated the step-by-step reasoning that humans use when they solve puzzles or make logical deductions. By the late 1980s and '90s, AI research had also developed highly successful methods for dealing with uncertain or incomplete information, employing concepts from probability and economics.

For difficult problems, most of these algorithms can require enormous computational resources — most experience a "combinatorial explosion": the amount of memory or computer time required becomes astronomical when the problem goes beyond a certain size. The search for more efficient problem-solving algorithms is a high priority for AI research.



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

Human beings solve most of their problems using fast, intuitive judgments rather than the conscious, step-by-step deduction that early AI research was able to model. AI has made some progress at imitating this kind of "sub-symbolic" problem solving: embodied agent approaches emphasize the importance of sensorimotor skills to higher reasoning; neural net research attempts to simulate the structures inside human and animal brains that give rise to this skill.

### **Importance of AI:-**

In order to maintain their competitiveness, companies feel compelled to adopt productivity increasing measures. Yet, they cannot relinquish the flexibility their production cycles need in order to improve their response, and thus, their positioning in the market. To achieve this, companies must combine these two seemingly opposed principles. Thanks to new technological advances, this combination is already a working reality in some companies. It is made possible today by the implementation of computer integrated manufacturing (CIM) and artificial intelligence (AI) techniques, fundamentally by means of expert systems (ES) and robotics. Depending on how these (AI/CIM) techniques contribute to automation, their immediate effects are an increase in productivity and cost reductions. Yet also, the system's flexibility allows for easier adaptation and, as a result, an increased ability to generate value, in other words, competitiveness is improved. The authors have analyzed three studies to identify the possible benefits or advantages, as well as the inconveniences, that this type of technique may bring to companies, specifically in the production field. Although the scope of the studies and their approach differ from one to the other, their joint contribution can be of unquestionable value in order to understand a little better the importance of ES within the production system

### **AI and its related field:-**

Robotics, Computer Vision, Image Processing, Voice recognition, Neural Networks, Some of the fields I can recollect right now are: Machine Learning Neural Networks Knowledge Representation (KR) Automated Planning , Scheduling , Learning and Reasoning Natural language processing Bayesian network Genetic Programming Ontology Search and optimization Epistemology Constraint satisfaction Natural Language Processing, Machine Consciousness, Computational Creativity, Robo-ethics, Pattern Recognition (including audio and visual recognition), Agents, Intelligent Tutoring Interfaces...the list goes on and on.

### **AI techniques:-**

Path finding is often associated with AI, because the A\* algorithm and many other path finding algorithms were developed by AI researchers. Several biology-inspired AI techniques are currently popular, and I receive questions about why I don't use them. Neural Networks model a brain learning by example given a set of right answers, it learns the general patterns. Reinforcement Learning models a brain learning by experience given some set of actions and an eventual reward or punishment, it learns which actions are good or bad. Genetic Algorithms model evolution by natural selection given some set of agents, let the better ones live and the worse ones die. Typically, genetic algorithms do not allow agents to learn during their lifetimes, while neural networks allow agents to learn only during their lifetimes. Reinforcement learning allows agents to learn during their lifetimes and share knowledge with other agents.

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



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

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

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

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

Instead, function approximation may be useful to construct components of pathfinding. It may be that the movement cost function is unknown. For example, the cost of moving across an orc-filled forest may not be known without actually performing the movement and fighting the battles. Using function approximation, each time the forest is crossed, the movement cost  $f(\text{number of orcs}, \text{size of forest})$  could be measured and fed into the neural network. For future path finding sessions, the new movement costs could be used to find better paths. Even when the function is unknown, function approximation is useful primarily when the function varies from game to game. If a single movement cost applies every time someone plays the game, the game developer can precompute it beforehand.

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

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

### Genetic Algorithms

Function approximation can be transformed into a function optimization problem. To find  $f(x)$  that approximates  $f(x)$ , set  $g(f) = \text{Sum of } (f(x) - f(x))^2 \text{ over all input } x$ .

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

Genetic Programming takes genetic algorithms a step further, and treats *programs* as the parameters. For example, you would be breeding path finding *algorithms* instead of *paths*, and your fitness function would



rate each algorithm based on how well it does. For path finding, we already have a good algorithm and we do not need to evolve a new one.

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

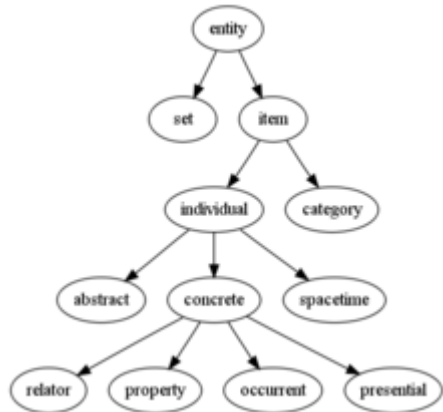
**Reinforcement Learning**

Like genetic algorithms, Reinforcement Learning is an unsupervised learning problem. However, unlike genetic algorithms, agents can learn during their lifetimes; it's not necessary to wait to see if they 'live' or 'die'. Also, it's possible for multiple agents experiencing different things to share what they've learned. Reinforcement learning has some similarities to the core of A\*. In A\*, reaching the end goal is propagated back to mark all the choices that were made along the path; other choices are discarded. In reinforcement learning, every state can be evaluated and its reward (or punishment) is propagated back to mark all the choices that were made leading up to that state. The propagation is made using a value function, which is somewhat like the heuristic function in A\*, except that it's updated as the agents try new things and learn what works. One of the key advantages of reinforcement learning and genetic algorithms over simpler approaches is that there is a choice made between *exploring* new things and *exploiting* the information learned so far. In genetic algorithms, the exploration is via mutation; in reinforcement learning, the exploration is via explicitly allowing the probability of choosing new actions. As with genetic algorithms, I don't believe reinforcement learning should be used for the path finding problem itself, but instead as a guide for teaching agents how to behave in the game world.

**Criteria for success:-**

If you visit our course "Artificial Intelligence. Success or failure of the humanity?" you will find out what the Artificial Intelligence is, how it appeared and what achievements have already been done in this field of knowledge. One of the most interesting parts of our course is the phenomena of indistinct linguistic concepts and processing of the information in conditions of uncertainty... sounds mysteriously doesn't it? :) And what's more important - you will get the opportunity to find out what positive and negative consequences the development of Artificial Intelligence is going to have! What's waiting for us - success or failure - you should decide by yourself....generate your opinion...choose your future!!!

Knowledge representation

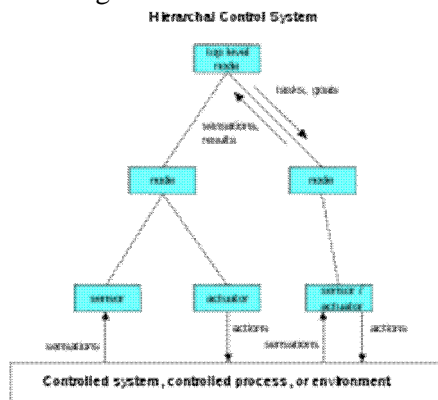


An ontology represents knowledge as a set of concepts within a domain and the relationships between those concepts.

Knowledge representation and knowledge engineering are central to AI research. Many of the problems machines are expected to solve will require extensive knowledge about the world. Among the things that AI needs to represent are: objects, properties, categories and relations between objects; situations, events, states and time; causes and effects; knowledge about knowledge (what we know about what other people know); and many other, less well researched domains. A representation of "what exists" is an ontology (borrowing a word from traditional philosophy), of which the most general are called upper ontologies.

Among the most difficult problems in knowledge representation are:

Planning



A hierarchical control system is a form of control system in which a set of devices and governing software is arranged in a hierarchy.

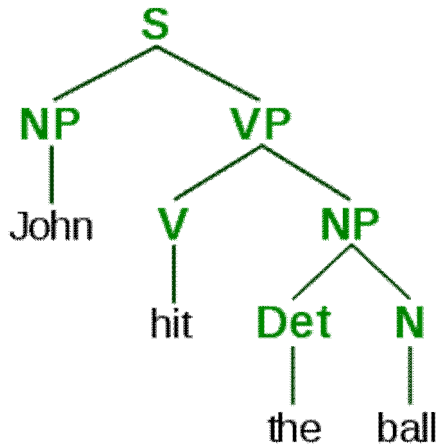
Intelligent agents must be able to set goals and achieve them. They need a way to visualize the future (they must have a representation of the state of the world and be able to make predictions about how their actions will change it) and be able to make choices that maximize the utility (or "value") of the available choices.

In classical planning problems, the agent can assume that it is the only thing acting on the world and it can be certain what the consequences of its actions may be. However, if this is not true, it must periodically check if the world matches its predictions and it must change its plan as this becomes necessary, requiring the agent to reason under uncertainty.

Learning

Machine learning has been central to AI research from the beginning. In 1956, at the original Dartmouth AI summer conference, Ray Solomonoff wrote a report on unsupervised probabilistic machine learning: "An Inductive Inference Machine". Unsupervised learning is the ability to find patterns in a stream of input. Supervised learning includes both classification and numerical regression. Classification is used to determine what category something belongs in, after seeing a number of examples of things from several categories. Regression is the attempt to produce a function that describes the relationship between inputs and outputs and predicts how the outputs should change as the inputs change. In reinforcement learning the agent is rewarded for good responses and punished for bad ones. These can be analyzed in terms of decision theory, using concepts like utility. The mathematical analysis of machine learning algorithms and their performance is a branch of theoretical computer science known as computational learning theory.

Natural language processing



A parse tree represents the syntactic structure of a sentence according to some formal grammar.

*Natural language processing*

Natural language processing gives machines the ability to read and understand the languages that humans speak. A sufficiently powerful natural language processing system would enable natural language user interfaces and the acquisition of knowledge directly from human-written sources, such as Internet texts. Some straightforward applications of natural language processing include information retrieval (or text mining) and machine translation.

**Problems:-**

- Use of symbolic reasoning .
- Focus on problems that do not respond to algorithmic solution (Heuristic) .
- Work on problems with inexact , missing , or poorly defined information .
- Provide answers that are sufficient but not exact .
- Deals with semantics as well as syntactic .
- Work with qualitative knowledge rather than quantitative knowledge .
- Use large amount of domain specific knowledge .

Comparison between intelligent computing and conventional computing:

	Intelligent Computing		Conventional Computing
1	Does not guarantee a solution to a given problem.	1	Guarantees a solution to a given problem.
2	Results may not be reliable and consistent	2	Results are consistent and reliable.
3	Programmer does not tell the system how to solve the given problem.	3	Programmer tells the system exactly how to solve the problem
4	Can solve a range of problems in a given domain.	4	Can solve only one problem at a time in a given domain

Applications of AI:

- Game playing :



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

Most games are played using a well-defined set of rules, this makes it easy to generate the search space and frees the researcher from many of the ambiguities and complexities inherent in less structured problems.

-Automated Reasoning & Theorem Proving :

Theorem provers function as intelligent assistants, letting human perform the more demanding tasks of decomposing a large problem into sub problems and devising heuristics for searching the space of possible proofs.

-Expert Systems :

Expert knowledge is a combination of a theoretical understanding of the problem and a collection of heuristic problem solving rules that experience has shown to be effective in the domain, expert systems are constructed by obtaining this knowledge from a human expert and coding it into a form that a computer may apply to similar problems.

*DENDRAL*, developed at Stanford in the late 1960, *DENDRAL* was designed to infer the structure of organic molecules from their chemical formulas and mass spectrographic information about the chemical bonds present in the molecules.

*MYCIN*, developed at Stanford in the mid\_1970, was one of the first programs to address the problems of reasoning with uncertain or incomplete information, *MYCIN* provided clear and logical explanation of its reasoning, used a control structure appropriate to the specific problem domain, and identified criteria to reliably evaluate its performance.

-Natural Language Understanding & Semantic Modeling:

One of the long-standing goals of artificial intelligence is the creation of programs that are capable of understanding human language, Not only does the ability to understand natural language seem to be one of the most fundamental aspects of human intelligence, but also its successful automation would have an incredible impact on the usability and effectiveness of computer systems.

-Planning & Robotics:

Planning is an important aspect of the effort to design Robots that perform their task with some degree of flexibility and responsiveness to the outside world, Briefly, Planning assumes a Robot that is capable of performing certain atomic actions, It attempts to find a sequence of those actions that will accomplish some higher-level task such as moving a cross an obstacle-filled room.

-Machine Learning:

The ability to learn is one of the most important components of intelligent behavior; an expert system may perform extensive and costly computations to solve a problem.

Unlike a human being, however, if it is given the same or a similar problem a second time, it will not remember the solution. It performs the same sequence of computation again.

This is true the second, third, fourth, and every time it solves that problem-hardly the behavior of an intelligent problem solver.

### **problem space and search: Defining the problem as a state space search:-**

State space search is a process used in the field of computer science, including artificial intelligence (AI), in which successive configurations or *states* of an instance are considered, with the goal of finding a *goal state* with a desired property.

Problems are often modeled as a state space, a set of *states* that a problem can be in. The set of states forms a graph where two states are connected if there is an *operation* that can be performed to transform the first state into the second.

State space search often differs from traditional computer science search methods because the state space is *implicit*: the typical state space graph is much too large to generate and store in memory. Instead, nodes are generated as they are explored, and typically discarded thereafter. A solution to a combinatorial search





तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
**Institute of Management & Technology**  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

instance may consist of the goal state itself, or of a path from some *initial state* to the goal state.

### Production system and its characteristics:-

A Knowledge representation formalism consists of collections of condition-action rules (Production Rules or Operators), a database which is modified in accordance with the rules, and a Production System Interpreter which controls the operation of the rules i.e The 'control mechanism' of a Production System, determining the order in which Production Rules are fired.

A system that uses this form of knowledge representation is called a production system. A production system consists of rules and factors. Knowledge is encoded in a declarative form which comprises of a set of rules of the form Situation ----- Action SITUATION that implies ACTION.

Example:-

IF the initial state is a goal state THEN quit.

The major components of an AI production system are

- i. A global database
- ii. A set of production rules and
- iii. A control system

The global database is the central data structure used by an AI production system. The production rules operate on the global database. Each rule has a precondition that is either satisfied or not by the database. If the precondition is satisfied, the rule can be applied. Application of the rule changes the database. The control system chooses which applicable rule should be applied and ceases computation when a termination condition on the database is satisfied. If several rules are to fire at the same time, the control system resolves the conflicts.

Four classes of production systems:-

1. A monotonic production system
2. A non monotonic production system
3. A partially commutative production system
4. A commutative production system.

Advantages of production systems:-

1. Production systems provide an excellent tool for structuring AI programs.
2. Production Systems are highly modular because the individual rules can be added, removed or modified independently.



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

3. The production rules are expressed in a natural form, so the statements contained in the knowledge base should be a recording of an expert thinking out loud.

#### **Issues in the design of the search :-**

One important disadvantage is the fact that it may be very difficult to analyse the flow of control within a production system because the individual rules don't call each other.

Production systems describe the operations that can be performed in a search for a solution to the problem. They can be classified as follows.

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

Partially commutative production system:-

A production system in which the application of a particular sequence of rules transforms state X into state Y, then any permutation of those rules that is allowable also transforms state x into state Y.

Theorem proving falls under monotonic partially commutative system. Blocks world and 8 puzzle problems like chemical analysis and synthesis come under monotonic, not partially commutative systems. Playing the game of bridge comes under non monotonic, not partially commutative system.

For any problem, several production systems exist. Some will be efficient than others. Though it may seem that there is no relationship between kinds of problems and kinds of production systems, in practice there is a definite relationship.

Partially commutative, monotonic production systems are useful for solving ignorable problems. These systems are important from a man implementation standpoint because they can be implemented without the ability to backtrack to previous states, when it is discovered that an incorrect path was followed. Such systems increase the efficiency since it is not necessary to keep track of the changes made in the search process.

#### **Problem Heuristic search techniques :Generate and test**

##### **Heuristic Search:**

Unlike Blind search Heuristic search algorithms use information regarding the problem to be solved thus attempting to reduce the search space. The ideal objective is to find the goal with least amount of efforts and time and to try to find the shortest path to the goal. Information regarding the problem is usually represented using an evaluation function. This evaluation function is used to compare between different states in a state space graph so that the best state is selected. The evaluation function is as follows:

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

where  $h(n)$  : the estimated distance from node (n) to the goal .  
 $g(n)$  : the distance between start node and node (n) .

##### Properties of a Heuristic Function

The behavior and efficiency of a search algorithm depends heavily on the accuracy and the properties of a



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

heuristic function  $h(n)$ . The heuristic function  $h(n)$  estimates the distance from a given node ( $n$ ) to the goal. Based on this estimate we can choose between different possible states and select the closest one to the goal (the one with the smaller  $h(n)$  ).

**Admissibility**

A search algorithm is admissible if it guarantees to find the minimal path to the goal if a such a path exists. An algorithm is considered admissible if it uses a heuristic function that is admissible.

A heuristic function  $h(n)$  is admissible if its estimated values are always less or equal to the real values of the distance to the goal. If we consider the estimated values  $h'(n)$  and the real values  $h(n)$  then:

$$h'(n) \leq h(n)$$

**Monotonicity**

A monotone heuristic function is one that guarantees that once a state is found (reached) it won't be found later in the search at a cheaper cost or a shorter path. In other words, the way (path) used to reach a given state is always guaranteed to be the best or lower cost path.

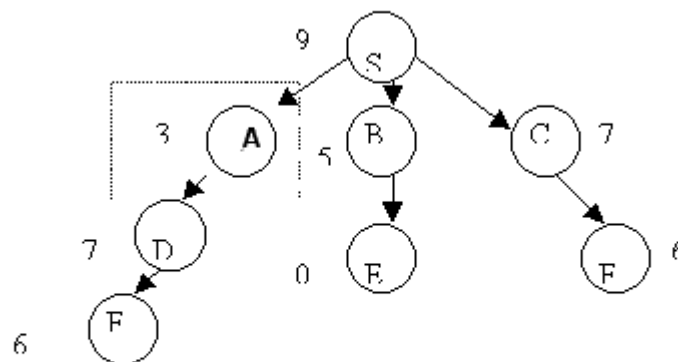
Characteristics of a heuristic function:

1- Under-Estimation: When a heuristic function gives a value to a node that is less than the actual value. It is called underestimation.  $h'(n) < h(n)$  where  $h'(n)$  is the estimated value and  $h(n)$  is the real value. A function which underestimates is still considered Admissible. It will find the shortest path to the goal, but it will waste or spend more time to find it.

Underestimation means that we make a node look better than its actually is.

consider this example:

### Under Estimation



The values of  $h(n)$  are correct for all the nodes except for node A it is underestimated. This value will make node A look good, and it will be expanded or selected before node B. However, when we reach node D we will find that this is the wrong path, and we will go back and select node B. What happened is that we wasted time

2- Over Estimation: is to assign a node in a state space graph a value that is larger than the actual value. It is to make a good node (state) look bad. Because this node looks bad, it will not be considered for expansion. The big problem is that if this node is on the path to the best goal, and there is another goal (not best), the search might find the second goal (not best) before finding the best goal.

**A function which overestimates is not Admissible.**

3- Perfect Estimation: If a heuristic function always gives consistent values to the nodes in the graph, then



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

we call it a perfect estimator. This means that the values assigned to the nodes always reflects the cost accurately.

*Heuristic Search Algorithms :*

1. Simple Hill-Climbing
2. Steepest Ascent Hill-Climbing
3. Best First Search A\* Algorithm
4. Min Max
5. Alpha-Beta Pruning

```
Begin
Open: = [Star];
Closed: = [ ];
While open! = [ ] do
  Begin
    Pop (open, X);
    If X is a goal then return (success)
    Else begin
      Expand X;
      Push (closed, X);
      Eliminate those children of X which are already on open or closed.
      Push (open, remaining children of X);
    End;
  End;
Return (failure);
End.
Procedure breadth_first_search;
Begin
Open: = [Star];
Closed: = [ ];
While open! = [ ] Do
  Begin
    Dequeue (open, X);
    If X is a goal then return (success)
    Else begin
      Expand X;
      Enqueue (closed, X);
      Eliminate those children of X which are already on open or closed.
      Enqueue (open, remaining children of X);
    End;
  End;
Return (failure);
End.
```



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
**Institute of Management & Technology**  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

Simple Hill-Climbing :

- 1) Evaluate the start state if goal then return (success) .  
else continue with start state as the current state .
- 2) Loop until a solution is found or until there are no new operator to apply to current node :
  - a) select a new operator and apply current state to produce a new state .
  - b) evaluate the new state :
    - i) if it is a goal then return (success) .
    - ii) if not goal but better than current state then make it the current state .
    - iii). if it is not better than current state then continue the loop .

Steepest Ascent Hill-Climbing :

- 1) Evaluate the initial state . if it is also a goal state , then return it and quit . otherwise , continue with the initial state as current state .
- 2) Loop until a solution is found or until a complete iteration produces no change to current state :
  - a) let SUCC be a state such that any possible successor of the current state will be better than SUCC .
  - b) for each operator that applies to the current state do :
    - i) apply the operator and generate a new state .
    - ii) evaluate the new state . if it is a goal state , return it and quit . If not , compare it to SUCC . if it is better , then set SUCC to this state . if it is not better , then leave SUCC alone .
  - c) if the SUCC is better than current state , then set current state to SUCC .

Heuristics in Game Playing

### 1. Full-Ply MinMax

To construct a complete plan for the entire state space. This plan shows all the possible moves allowed and shows the winning and losing states.

Assumptions:

Usually there are two players, the first is called Max and the second is called Min.

We will assume that Max is the computer and Min is the person playing against the computer. Our job is to help Max win the game (any two player game) by writing a program using the MinMax algorithm. We will assume the following:

1. The person playing against the computer is a professional player, who does not make mistakes.
2. we will not choose any move that has any chances of losing the game.
- 3.

Steps:

1. Generate all possible moves (states) in a depth-first fashion. A graph is created.
2. Mark each level in the graph with either MAX or MIN in an ordered fashion.
3. When you reach a leaf node. It means that you reached a winning, losing, or Tie state for either MAX or MIN depending on the Mark of that level.
4. Assign a winning state for MAX the value 1 and a losing state for MAX a 0. If it was a winning state for MIN assign it a value of 0.



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

- propagate the values assigned to leaf nodes, to the upper levels until reaching the root node.

Heuristic search techniques: Generate and test, hill climbing, best first search technique, Problem reduction, constraint satisfaction

### Search Algorithms in AI

Search techniques are general problem-solving methods. When there is a formulated search problem, a set of states, a set of operators, an initial state, and a goal criterion we can use search techniques to solve the problem (Pearl & Korf, 1987). A search algorithm is explained in simple terms by Cawsey (1998): Suppose that you are trying to find your way around in a small town, which has many one way streets. Your initial state (A) to the target destination (B) there are many routes, but you do not know which ones to take. You can try as many in real life, although it might be a difficult process. There might be much inconvenience in the whole picture: Once you have passed through one street you may not be able to go back, as it is one way, or you may be stuck in a park. This simplified idea of searching for a place indicates that we can actually map out all the possible routes in such an example; even it is exhausting you eventually find a way. In small scale search problems, as introduced above, simple search techniques are sufficient to do systematic search. However, sometimes we need systematically search for such a route, for instance, on a much more complex map. How do we do that? Search algorithms are good for solving such problems. The problems could refer to physical problems such as walking or driving from A to B; or it could be abstract actions such as in a set of steps in a theorem, which will allow us to prove from a set of facts.

### Generate-And-Test Algorithm:-

Generate-and-test search algorithm is a very simple algorithm that guarantees to find a solution if done systematically and there exists a solution.

Algorithm: Generate-And-Test

- 1.Generate a possible solution.
- 2.Test to see if this is the expected solution.
- 3.If the solution has been found quit else go to step 1.

Potential solutions that need to be generated vary depending on the kinds of problems. For some problems the possible solutions may be particular points in the problem space and for some problems, paths from the start state.

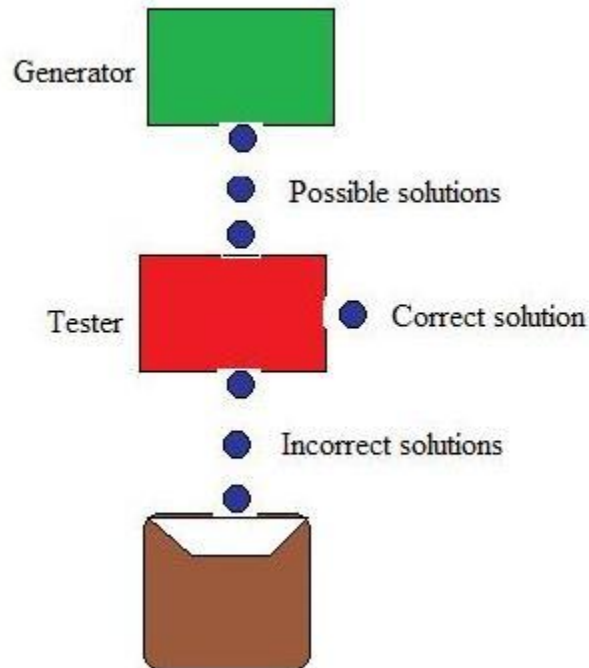


Figure: Generate And Test

Generate-and-test, like depth-first search, requires that complete solutions be generated for testing. In its most systematic form, it is only an exhaustive search of the problem space. Solutions can also be generated randomly but solution is not guaranteed. This approach is what is known as British Museum algorithm: finding an object in the British Museum by wandering randomly.

#### Systematic Generate-And-Test

While generating complete solutions and generating random solutions are the two extremes there exists another approach that lies in between. The approach is that the search process proceeds systematically but some paths that unlikely to lead the solution are not considered. This evaluation is performed by a heuristic function.

Depth-first search tree with backtracking can be used to implement systematic generate-and-test procedure. As per this procedure, if some intermediate states are likely to appear often in the tree, it would be better to modify that procedure to traverse a graph rather than a tree.

#### Generate-And-Test And Planning

Exhaustive generate-and-test is very useful for simple problems. But for complex problems even heuristic generate-and-test is not very effective technique. But this may be made effective by combining with other techniques in such a way that the space in which to search is restricted. An AI program DENDRAL, for example, uses plan-Generate-and-test technique. First, the planning process uses constraint-satisfaction techniques and creates lists of recommended and contraindicated substructures. Then the generate-and-test procedure uses the lists generated and required to explore only a limited set of structures. Constrained in this way, generate-and-test proved highly effective. A major weakness of planning is that it often produces inaccurate solutions as there is no feedback from the world. But if it is used to produce only pieces of solutions then lack of detailed accuracy becomes unimportant.



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

Best-first search is a search algorithm which explores a graph by expanding the most promising node chosen according to a specified rule.

Judea Pearl described best-first search as estimating the promise of node  $n$  by a "heuristic evaluation function  $f(n)$  which, in general, may depend on the description of  $n$ , the description of the goal, the information gathered by the search up to that point, and most important, on any extra knowledge about the problem domain."

Some authors have used "best-first search" to refer specifically to a search with a heuristic that attempts to predict how close the end of a path is to a solution, so that paths which are judged to be closer to a solution are extended first. This specific type of search is called greedy best-first search.

Efficient selection of the current best candidate for extension is typically implemented using a priority queue.

The A\* search algorithm is an example of best-first search. Best-first algorithms are often used for path finding in combinatorial search.

#### Algorithm

OPEN = [initial state]

while OPEN is not empty

do

1. Remove the best node from OPEN, call it  $n$ .
2. If  $n$  is the goal state, backtrace path to  $n$  (through recorded parents) and return path.
3. Create  $n$ 's successors.
4. Evaluate each successor, add it to OPEN, and record its parent.

done

Note that this version of the algorithm is not *complete*, i.e. it does not always find a possible path between two nodes even if there is one. For example, it gets stuck in a loop if it arrives at a dead end, that is a node with the only successor being its parent. It would then go back to its parent, add the dead-end successor to the OPEN list again, and so on.

The following version extends the algorithm to use an additional CLOSED list, containing all nodes that have been evaluated and will not be looked at again. As this will avoid any node being evaluated twice, it is not subject to infinite loops.

OPEN = [initial state]

CLOSED = []

while OPEN is not empty

do

1. Remove the best node from OPEN, call it  $n$ , add it to CLOSED.
2. If  $n$  is the goal state, backtrace path to  $n$  (through recorded parents) and return path.
3. Create  $n$ 's successors.
4. For each successor do:
  - a. If it is not in CLOSED: evaluate it, add it to OPEN, and record its parent.
  - b. Otherwise: change recorded parent if this new path is better than previous one.

done

Also note that the given pseudo code of both versions just terminates when no path is found. An actual implementation would of course require special handling of this case.

#### **Hill Climbing:-**

In computer science, hill climbing is a mathematical optimization technique which belongs to the family of local search. It is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts





तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

to find a better solution by incrementally changing a single element of the solution. If the change produces a better solution, an incremental change is made to the new solution, repeating until no further improvements can be found.

For example, hill climbing can be applied to the travelling salesman problem. It is easy to find an initial solution that visits all the cities but will be very poor compared to the optimal solution. The algorithm starts with such a solution and makes small improvements to it, such as switching the order in which two cities are visited. Eventually, a much shorter route is likely to be obtained.

Hill climbing is good for finding a local optimum (a solution that cannot be improved by considering a neighbouring configuration) but it is not guaranteed to find the best possible solution (the global optimum) out of all possible solutions (the search space). The characteristic that only local optima are guaranteed can be cured by using restarts (repeated local search), or more complex schemes based on iterations, like iterated local search, on memory, like reactive search optimization and tabu search, or memory-less stochastic modifications, like simulated annealing.

The relative simplicity of the algorithm makes it a popular first choice amongst optimizing algorithms. It is used widely in artificial intelligence, for reaching a goal state from a starting node. Choice of next node and starting node can be varied to give a list of related algorithms. Although more advanced algorithms such as simulated annealing or tabu search may give better results, in some situations hill climbing works just as well. Hill climbing can often produce a better result than other algorithms when the amount of time available to perform a search is limited, such as with real-time systems. It is an anytime algorithm: it can return a valid solution even if it's interrupted at any time before it ends.

### Greedy BFS

Using a greedy algorithm, expand the first successor of the parent. After a successor is generated:

1. If the successor's heuristic is better than its parent, the successor is set at the front of the queue (with the parent reinserted directly behind it), and the loop restarts.
2. Else, the successor is inserted into the queue (in a location determined by its heuristic value). The procedure will evaluate the remaining successors (if any) of the parent.

### Best-First search:

Begin

Open := [Start];

Closed := [ ];

While open (priority queue) != [ ] do

  Begin

    Dequeue (open ,X);

    If X=goal THEN return the path from S to X and EXIT.

    Else begin

      Expand X;

      FOR each child of X do:

        Case:

        [1] The child is not on open or closed.

          Begin

          Assign the child a heuristic value.

          Enqueue (open, Child);

          End;

        [2] The child is already on open;



```

        if the child was reached by a shorter path THEN
            assign the state on open the shorter path.
    [3] The child is already on closed;
        if the child was reached by a shorter path.
            Begin
                Remove the child from closed;
                Enqueue (open, child);
            End;
        End;
    Enqueue (closed, X);
    SORT the queue;
End;
Return fail;
End.

```

### A\* Search Algorithm

Heuristics search makes use of the fact that most problem spaces provide some information that distinguishes among states in terms of their likelihood of leading to a goal. This information is called a heuristic evaluation function (Pearl & Korf, 1987). In other words, the goal of a heuristic search is to reduce the number of nodes searched in seeking a goal (Kopec & Marsland, 2001).

Most widely used best first search form is called A\*, which is pronounced as A star. It is a heuristic searching method, and used to minimize the search cost in a given problem (Bolc & Cytowski, 1992). It aims to find the least-cost path from a given initial node to the specific goal. It is an extended form of best-first search algorithm. Best first-search algorithm tries to find a solution to minimize the total cost of the search pathway, too. However, the difference from Best-First Search is that A\* also takes into account the cost from the start, and not simply the local cost from the previously been node. Best-first search finds a goal state in any predetermined problem space. However, it cannot guarantee that it will choose the shortest path to the goal (Pearl & Korf, 1987). For instance, if there are two options to chose from, one of which is a long way from the initial point but has a slightly shorter estimate of distance to the goal, and another that is very close to the initial state but has a slightly longer estimate of distance to the goal, best-first search will always choose to expand next the state with the shorter estimate. The A\* algorithm fixes the best first search's this particular drawback (Pearl & Korf, 1987).

In short, A\* algorithm searches all possible routes from a starting point until it finds the shortest path or cheapest cost to a goal. The terms like shortest path, cheapest cost here refer to a general notion. It could be some other alternative term depending on the problem. For instance, in a map problem the cost is replaced by the term distance (Cawsey, 1998). This may reduce the necessity to search all the possible pathways in a search space, and result in faster solution. A\* evaluates nodes by combining  $g(n)$  and  $h(n)$ . In the standard terminology used when talking about A\*:

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

The purpose of this equation is to obtain the lowest  $f$  score in a given problem.  $n$  being node number crossed until the final node,

$f(n)$  is the total search cost,  $g(n)$  is actual lowest cost (shortest distance traveled) of the path from initial start point to the node  $n$ ,  $h(n)$  is the estimated of cost of cheapest (distance) from the node  $n$  to a goal node. This part of the equation is also called heuristic function/estimation.



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

At each node, the lowest  $f$  value is chosen to be the next step to expand until the goal node is chosen and reached for expansion. (Pearl & Korf, 1987). Whenever the heuristic function satisfies certain conditions,  $A^*$  search is both complete and optimal (Russell & Norvig, 2003).

Characteristics of  $A^*$  Search Algorithm

*Admissibility.* Strategies which guarantee optimal solution, if there is any solution, are called admissible. There are few items which are needed to be satisfied for  $A^*$  to be admissible. If a tree-search is in use, the optimality of  $A^*$  is straightforward to be analyzed (Russell & Norvig, 2003). In this case,  $A^*$  is optimal, if  $h(n)$  is an admissible heuristic.

$h^*(n)$  = the true minimal cost to goal from  $n$ . A heuristic  $h$  is admissible if  $h(n) \leq h^*(n)$  for all states  $n$ .

*Convergence.* A search strategy is convergent if it promises finding a path, a solution graph, or information if they exist (Bolc & Cytowski, 1992). If there is a solution,  $A^*$  will always find a solution. The convergence properties of  $A^*$  search algorithm are satisfied for any network with a non-negative cost function, either finite or infinite. For a finite network with a non-negative cost function, If  $A^*$  terminates after finding a solution, or if there is no solution, then it is convergent. The number of the paths in a cyclic path is finite. A node which is previously examined node, lets say  $w$ , is revisited only if the search finds a smaller cost than the previous one. The cost function is non-negative; therefore an edge can be examined only once. Thus,  $A^*$  is convergent (Bolc & Cytowski, 1992).

Problem with  $A^*$  Search Algorithm

According to Pearl & Korf (1987) the main shortcoming of  $A^*$ , and any best-first search, is its memory requirement. Because the entire open pathway list must be saved,  $A^*$  is space-limited in practice and is no more practical than breadth first search. For large search spaces,  $A^*$  will run out of memory. Fortunately, a mass of memory could be saved with  $IDA^*$  (Iterative Deepening  $A^*$ ).

Pseudo-code  $A^*$

(Anonymous, 2006)

Create the open list of nodes, initially containing only our starting node

Create the closed list of nodes, initially empty

```
while (we have not reached our goal) {
    consider the best node in the open list (the node with the lowest  $f$  value)

    if (this node is the goal) {
        then we're done
    }
    else {
        move the current node to the closed list and consider all of its neighbors

        for (each neighbor) {
            if (this neighbor is in the closed list and our current  $g$  value is lower) {
                update the neighbor with the new, lower,  $g$  value
                change the neighbor's parent to our current node
            }
            else if (this neighbor is in the open list and our current  $g$  value is lower) {
                update the neighbor with the new, lower,  $g$  value
                change the neighbor's parent to our current node
            }
            else this neighbor is not in either the open or closed list {
```

add the neighbor to the open list and set its g value

```

    }
  }
}

```

**Problem Reduction with AO\* Algorithm:-**

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 arc 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.

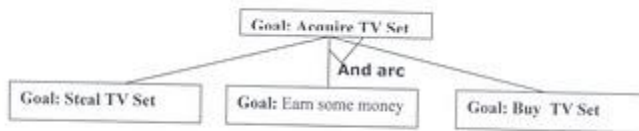


Figure shows AND - Or graph - an example.

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.

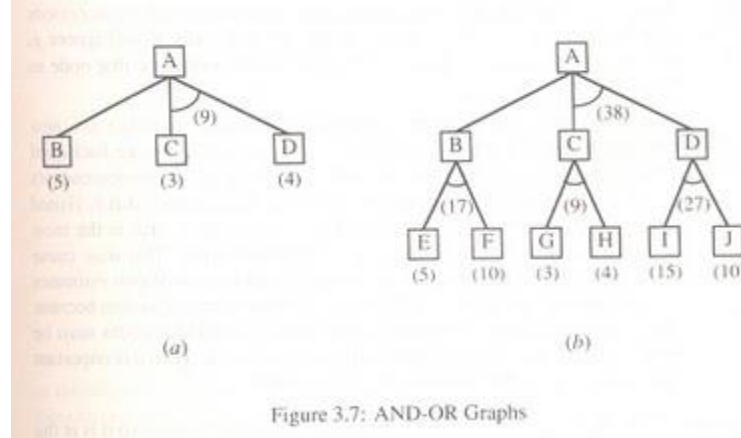


Figure 3.7: AND-OR Graphs

FIGURE : AND - OR graph

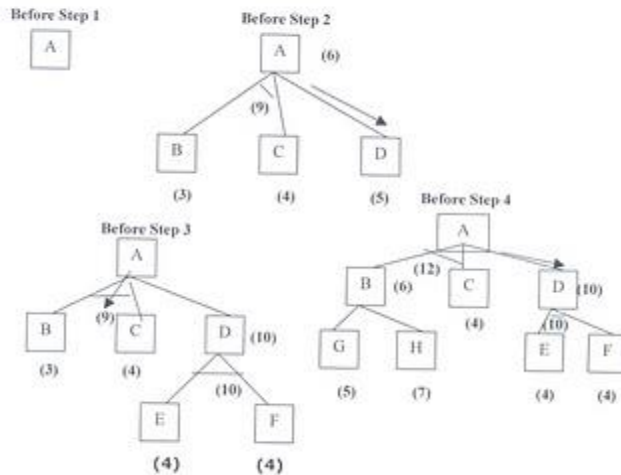
In figure (a) the top node A has been expanded producing two area one leading to B and leading to C-D . the numbers at each node represent the value of f ' at that node (cost of getting to the goal state from current state). For simplicity, it is assumed that every operation(i.e. applying a rule) has unit cost, i.e., each arc 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 on a value but also on whether that node is part of the current best path from the initial node. 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 best 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 compute  $f'$  (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 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



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

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 expensive to be practical.

For representing above graphs AO\* algorithm is as follows

#### AO\* ALGORITHM:

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

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

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

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

(a) add SUCCESSOR to graph G

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

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

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

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

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

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

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

(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.



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

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.

#### Constraint satisfaction

In artificial intelligence and operations research, constraint satisfaction is the process of finding a solution to a set of constraints that impose conditions that the variables must satisfy. A solution is therefore a vector of variables that satisfies all constraints.

The techniques used in constraint satisfaction depend on the kind of constraints being considered. Often used are constraints on a finite domain, to the point that constraint satisfaction problems are typically identified with problems based on constraints on a finite domain. Such problems are usually solved via search, in particular a form of backtracking or local search. Constraint propagation are other methods used on such problems; most of them are incomplete in general, that is, they may solve the problem or prove it unsatisfiable, but not always. Constraint propagation methods are also used in conjunction with search to make a given problem simpler to solve. Other considered kinds of constraints are on real or rational numbers; solving problems on these constraints is done via variable elimination or the simplex algorithm.

Constraint satisfaction originated in the field of artificial intelligence in the 1970s (see for example (Laurière 1978)). During the 1980s and 1990s, embedding of constraints into a programming language were developed. Languages often used for constraint programming are Prolog and C++.

#### **Constraint satisfaction problem:-**

As originally defined in artificial intelligence, constraints enumerate the possible values a set of variables may take. Informally, a finite domain is a finite set of arbitrary elements. A constraint satisfaction problem on such domain contains a set of variables whose values can only be taken from the domain, and a set of constraints, each constraint specifying the allowed values for a group of variables. A solution to this problem is an evaluation of the variables that satisfies all constraints. In other words, a solution is a way for assigning a value to each variable in such a way that all constraints are satisfied by these values.

In some circumstances, there may exist additional requirements: one may be interested not only in the solution (and in the fastest or most computationally efficient way to reach it) but in how it was reached; e.g. one may want the "simplest" solution ("simplest" in a logical, non computational sense that has to be precisely defined). This is often the case in logic games such as Sudoku. In practice, constraints are often expressed in compact form, rather than enumerating all values of the variables that would satisfy the



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
**Institute of Management & Technology**  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

constraint. One of the most used constraints is the one establishing that the values of the affected variables must be all different. Problems that can be expressed as constraint satisfaction problems are the Eight queens puzzle, the Sudoku solving problem, the Boolean satisfiability problem, scheduling problems and various problems on graphs such as the graph coloring problem. While usually not included in the above definition of a constraint satisfaction problem, arithmetic equations and inequalities bound the values of the variables they contain and can therefore be considered a form of constraints. Their domain is the set of numbers (either integer, rational, or real), which is infinite: therefore, the relations of these constraints may be infinite as well; for example,  $X = Y + 1$  has an infinite number of pairs of satisfying values. Arithmetic equations and inequalities are often not considered within the definition of a "constraint satisfaction problem", which is limited to finite domains. They are however used often in constraint programming.

**Solving Constraint satisfaction problems on finite domains are typically solved using a form of search. The most used techniques are variants of backtracking, constraint propagation, and local search. These techniques are used on problems with nonlinear constraints.**

In case there is a requirement on "simplicity", a pure logic, pattern based approach was first introduced for the Sudoku CSP in the book *The Hidden Logic of Sudoku*. It has recently been generalized to any finite CSP in another book by the same author: *Constraint Resolution Theories*.

Variable elimination and the simplex algorithm are used for solving linear and polynomial equations and inequalities, and problems containing variables with infinite domain. These are typically solved as optimization problems in which the optimized function is the number of violated constraints.

Complexity

Solving a constraint satisfaction problem on a finite domain is an NP complete problem with respect to the domain size. Research has shown a number of tractable subcases, some limiting the allowed constraint relations, some requiring the scopes of constraints to form a tree, possibly in a reformulated version of the problem. Research has also established relationship of the constraint satisfaction problem with problems in other areas such as finite model theory.

A very different aspect of complexity appears when one fixes the size of the domain. It is about the complexity distribution of minimal instances of a CSP of fixed size (e.g. Sudoku(9x9)). Here, complexity is measured according to the above-mentioned "simplicity" requirement (see *Unbiased Statistics of a CSP*

## UNIT - II

### **Knowledge representation: Definition and importance of knowledge**

#### **Definition:-**

Knowledge is, roughly, useful information. It is information that's adapted to a purpose. It is good explanations, and it is solutions to problems people had. Knowledge shouldn't be expected to be perfect. A partial solution is still knowledge, even if it contains some mistakes, and can be improved on in the future.

Knowledge is created by imaginative and critical thought. The key ingredients are both creativity and criticism. We need numerous ideas, including ones that aren't obvious. And we need error correction to get rid of flaws. With those two components, we can improve our knowledge and learn new things.

It's also important to be sensitive to problems. Problems are opportunities to learn something new, and to improve our lives. But some people are scared of problems, or consider problems inevitable and permanent. "Life isn't perfect, and who do you think you are trying to do better than thousands of smart people before you? Some problems are never going to go away, and you should just get accustomed to them." These people don't notice, keep track of, and make an effort to solve problems as well as they





तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

could with a better attitude. That means they solve fewer problems, and correct fewer of the problem-causing flaws in their ideas.

### **Importance of Knowledge:-**

Problems *can* be solved and knowledge *can* be created. What is there to stop us? There are the laws of physics. We can never make a perpetual motion machine. And there's our preferences. If we don't want to solve a problem, we won't. And there's our knowledge. If we don't know enough about a problem then we may have to learn more before we solve it.

None of these obstacles should ever make us unhappy. It's possible to have a great life without violating the laws of physics. It's possible to have a productive, happy life without knowing everything: just work on accessible problems and make progress. If we want to live in this way, we won't be upset when we don't solve some other irrelevant problem.

Most people think that knowledge is justified, true belief. This is an authoritarian conception of knowledge, and a perfectionist conception. It insists that if an idea is only a partial solution, or contains a mistake, then it's not genuine knowledge. And it encourages appeals to authority which serve as justifications.

If a person believes that he has a justified, *true* belief then he has no reason to listen to criticism of his belief, or to listen to dissenting opinions. Any idea which contradicts a true belief must be false. Therefore, all criticism is irrelevant, and anyone who disagrees is mistaken. The only thing to do is educate them, not debate with them, and not consider that they might be right and we might be able to mutually learn from each other. Confidence that one definitely knows the final truth leaves one with no reason to try to correct errors; it's actually foolish.

*Justification* is a chimera. Suppose I justify an idea with a justification  $J_1$ . Now let's consider  $J_1$ .  $J_1$  is itself an idea. And it might be mistaken, so we'll need to justify  $J_1$  too. So we do: we think of a justification of  $J_1$  which we'll call  $J_2$ . Unfortunately, this leads to the same problem again. How do we know  $J_2$  is correct? So we offer  $J_3$ , and then  $J_4$ , and so on. And every single justification is critically important. If  $J_3$  isn't correct or isn't justified, then neither is  $J_2$ . And if  $J_2$  isn't, then  $J_1$  isn't justified. And that would mean the original idea isn't either, which would mean it's not knowledge according to the justified, true belief conception of knowledge.

As you can see, justification leads to an infinite regress. There is no end to the justifications needed. There are two ways to try to get out of this problem. The first is circular. You use  $J_2$  to justify  $J_4$ , say. But circular arguments are invalid. The second way out is to declare some ideas to be self-evident or self-justifying. When you get to them, you just stop, and you never consider if they might be mistaken. This is circular too. It's justifying  $J_4$  using  $J_4$ . Further, how do you decide which propositions are self-justifying? You'll need an idea about that, and it needs to be correct, so you better justify it. No problem has been solved.

The way out of this mess is to stop seeking justifications at all. Instead, we can pursue knowledge as I describe it above: imperfect but useful ideas, which we don't claim are justified, but we do improve as much as we can, and remove as many errors as we can from. In this way, our knowledge is our best ideas so far. What's wrong with that?

A common question is if we don't accept justifications, then how can we ever take practical action when we don't have a justified, true belief about which action is best. This is easy. We should act on our best ideas. What else would we do? Act on ideas we consider inferior?

Using the justificationist approach, when we consider a new idea the main question asked is, "How do we know this is true? How can we support or prove it?" Support and proof are just different words for "justify", and have the same problems I described.



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

When we consider a new idea, the main question should be: "Do you (or anyone else) see anything wrong with it? And do you (or anyone else) have a better idea?" If the answers are 'no' and 'no' then we can accept it as our best idea for now.

### **Knowledge representation:-**

Knowledge representation (KR) is an area of artificial intelligence research aimed at representing knowledge in symbols to facilitate inferencing from those knowledge elements, creating new elements of knowledge. The KR can be made to be independent of the underlying knowledge model or knowledge base system (KBS) such as a semantic network.

Knowledge Representation (KR) research involves analysis of how to accurately and effectively reason and how best to use a set of symbols to represent a set of facts within a knowledge domain. A symbol vocabulary and a system of logic are combined to enable inferences about elements in the KR to create new KR sentences. Logic is used to supply formal semantics of how reasoning functions should be applied to the symbols in the KR system. Logic is also used to define how operators can process and reshape the knowledge. Examples of operators and operations include, negation, conjunction, adverbs, adjectives, quantifiers and modal operators. The logic is interpretation theory. These elements--symbols, operators, and interpretation theory--are what give sequences of symbols meaning within a KR.

A key parameter in choosing or creating a KR is its *expressivity*. The more expressive a KR, the easier and more compact it is to express a fact or element of knowledge within the semantics and grammar of that KR. However, more expressive languages are likely to require more complex logic and algorithms to construct equivalent inferences. A highly expressive KR is also less likely to be complete and consistent. Less expressive KR's may be both complete and consistent. Auto epistemic temporal modal logic is a highly expressive KR system, encompassing meaningful chunks of knowledge with brief, simple symbol sequences (sentences). Propositional logic is much less expressive but highly consistent and complete and can efficiently produce inferences with minimal algorithm complexity. Nonetheless, only the limitations of an underlying knowledge base affect the ease with which inferences may ultimately be made (once the appropriate KR has been found). This is because a knowledge set may be exported from a knowledge model or knowledge base system (KBS) into different KR's, with different degrees of expressiveness, completeness, and consistency. If a particular KR is inadequate in some way, that set of problematic KR elements may be transformed by importing them into a KBS, modified and operated on to eliminate the problematic elements or augmented with additional knowledge imported from other sources, and then exported into a different, more appropriate KR.

A knowledge representation (KR) is most fundamentally a surrogate, a substitute for the thing itself, used to enable an entity to determine consequences by thinking rather than acting, i.e., by reasoning about the world rather than taking action in it.

- It is a set of ontological commitments, i.e., an answer to the question: In what terms should I think about the world?
- It is a fragmentary theory of intelligent reasoning, expressed in terms of three components: (i) the representation's fundamental conception of intelligent reasoning; (ii) the set of inferences the representation sanctions; and (iii) the set of inferences it recommends.
- It is a medium for pragmatically efficient computation, i.e., the computational environment in which thinking is accomplished. One contribution to this pragmatic efficiency is supplied by the



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

guidance a representation provides for organizing information so as to facilitate making the recommended inferences.

- It is a medium of human expression, i.e., a language in which we say things about the world."

### **Issues in Knowledge representation:-**

Some issues that arise in knowledge representation from an AI perspective are:

- How do people represent knowledge?
- What is the nature of knowledge?
- Should a representation scheme deal with a particular domain or should it be general purpose?
- How expressive is a representation scheme or formal language?
- Should the scheme be declarative or procedural?

There has been very little top-down discussion of the knowledge representation (KR) issues and research in this area is a well aged quillwork. There are well known problems such as "spreading activation" (this is a problem in navigating a network of nodes), "sub assumption" (this is concerned with selective inheritance; e.g. an ATV can be thought of as a specialization of a car but it inherits only particular characteristics) and "classification." For example a tomato could be classified both as a fruit and a vegetable.

In the field of artificial intelligence, problem solving can be simplified by an appropriate choice of *knowledge representation*. Representing knowledge in some ways makes certain problems easier to solve. For example, it is easier to divide numbers represented in Hindu-Arabic numerals than numbers represented as Roman numerals.

### Characteristics

A good knowledge representation covers six basic characteristics:

- Coverage, which means the KR covers a breadth and depth of information. Without a wide coverage, the KR cannot determine anything or resolve ambiguities.
- Understandable by humans. KR is viewed as a natural language, so the logic should flow freely. It should support modularity and hierarchies of classes (Polar bears are bears, which are animals). It should also have simple primitives that combine in complex forms.
- Consistency. If John closed the door, it can also be interpreted as the door was closed by John. By being consistent, the KR can eliminate redundant or conflicting knowledge.
- Efficient
- Easiness for modifying and updating.
- Supports the intelligent activity which uses the knowledge base

To gain a better understanding of why these characteristics represent a good knowledge representation, think about how an encyclopedia (e.g. Wikipedia) is structured. There are millions of articles (coverage), and they are sorted into categories, content types, and similar topics (understandable). It redirects different titles but same content to the same article (consistency). It is efficient, easy to add new pages or update existing ones, and allows users on their mobile phones and desktops to view its knowledge base.

### Language and notation

Some think it is best to represent knowledge in the same way that it is represented in the human mind, or to represent knowledge in the form of human language.

Psycholinguistics investigates how the human mind stores and manipulates language. Other branches of cognitive science examine how human memory stores sounds, sights, smells, emotions, procedures, and



abstract ideas. Science has not yet completely described the internal mechanisms of the brain to the point where they can simply be replicated by computer programmers.

Various<sup>1</sup> artificial languages and notations have been proposed for representing knowledge. They are typically based on logic and mathematics, and have easily parsed grammars to ease machine processing. They usually fall into the broad domain of ontologies.

### Using Predicate Logic : Representing Simple Facts in logic

Over the past 100-odd years, what is now called first-order logic has gone by many names, including: first-order predicate calculus, the lower predicate calculus, quantification theory, and predicate logic (a less precise term). First-order logic is distinguished from propositional logic by its use of quantified variables. First-order logic with a specified domain of discourse over which the quantified variables range, one or more interpreted predicate letters, and proper axioms involving the interpreted predicate letters, is a first-order theory.

The adjective "first-order" distinguishes first-order logic from higher-order logic in which there are predicates having predicates or functions as arguments, or in which one or both of predicate quantifiers or function quantifiers are permitted. In first-order theories, predicates are often associated with sets. In interpreted higher-order theories, predicates may be interpreted as sets of sets.

#### Introduction

A predicate resembles a function that returns either True or False. Consider the following sentences: "Socrates is a philosopher", "Plato is a philosopher". In propositional logic these are treated as two unrelated propositions, denoted for example by  $p$  and  $q$ . In first-order logic, however, the sentences can be expressed in a more parallel manner using the predicate  $\text{Phil}(a)$ , which asserts that the object represented by  $a$  is a philosopher. Thus if  $a$  represents Socrates then  $\text{Phil}(a)$  asserts the first proposition,  $p$ ; if  $a$  instead represents Plato then  $\text{Phil}(a)$  asserts the second proposition,  $q$ . A key aspect of first-order logic is visible here: the string "Phil" is a syntactic entity which is given semantic meaning by declaring that  $\text{Phil}(a)$  holds exactly when  $a$  is a philosopher. An assignment of semantic meaning is called an interpretation.

First-order logic allows reasoning about properties that are shared by many objects, through the use of variables. For example, let  $\text{Phil}(a)$  assert that  $a$  is a philosopher and let  $\text{Schol}(a)$  assert that  $a$  is a scholar. Then the formula

$$\text{Phil}(a) \rightarrow \text{Schol}(a)$$

asserts that if  $a$  is a philosopher then  $a$  is a scholar. The symbol  $\rightarrow$  is used to denote a conditional (if/then) statement. The hypothesis lies to the left of the arrow and the conclusion to the right. The truth of this formula depends on which object is denoted by  $a$ , and on the interpretations of "Phil" and "Schol".

Assertions of the form "for every  $a$ , if  $a$  is a philosopher then  $a$  is a scholar" require both the use of variables and the use of a quantifier. Again, let  $\text{Phil}(a)$  assert  $a$  is a philosopher and let  $\text{Schol}(a)$  assert that  $a$  is a scholar. Then the first-order sentence

$$\forall a(\text{Phil}(a) \rightarrow \text{Schol}(a))$$

asserts that no matter what  $a$  represents, if  $a$  is a philosopher then  $a$  is scholar. Here  $\forall$ , the universal quantifier, expresses the idea that the claim in parentheses holds for *all* choices of  $a$ .

To show that the claim "If  $a$  is a philosopher then  $a$  is a scholar" is false, one would show there is some philosopher who is not a scholar. This counterclaim can be expressed with the existential quantifier  $\exists$ :

$$\exists a(\text{Phil}(a) \wedge \neg \text{Schol}(a)).$$

Here:

- $\neg$  is the negation operator:  $\neg \text{Schol}(a)$  is true if and only if  $\text{Schol}(a)$  is false, in other words if and only if  $a$  is not a scholar.



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

- $\wedge$  is the conjunction operator:  $\text{Phil}(a) \wedge \neg\text{Schol}(a)$  asserts that  $a$  is a philosopher and also not a scholar.

The predicates  $\text{Phil}(a)$  and  $\text{Schol}(a)$  take only one parameter each. First-order logic can also express predicates with more than one parameter. For example, "there is someone who can be fooled every time" can be expressed as:

$$\exists x(\text{Person}(x) \wedge \forall y(\text{Time}(y) \rightarrow \text{Canfool}(x, y))).$$

Here  $\text{Person}(x)$  is interpreted to mean  $x$  is a person,  $\text{Time}(y)$  to mean that  $y$  is a moment of time, and  $\text{Canfool}(x,y)$  to mean that (person)  $x$  can be fooled at (time)  $y$ . For clarity, this statement asserts that there is at least one person who can be fooled at all times, which is stronger than asserting that at all times at least one person exists who can be fooled. Asserting the latter (that there is always at least one foolable person) does not signify whether this foolable person is always the same for all moments of time.

The range of the quantifiers is the set of objects that can be used to satisfy them. (In the informal examples in this section, the range of the quantifiers was left unspecified.) In addition to specifying the meaning of predicate symbols such as  $\text{Person}$  and  $\text{Time}$ , an interpretation must specify a nonempty set, known as the domain of discourse or universe, as a range for the quantifiers. Thus a statement of the form  $\exists a\text{Phil}(a)$  is said to be true, under a particular interpretation, if there is some object in the domain of discourse of that interpretation that satisfies the predicate that the interpretation uses to assign meaning to the symbol  $\text{Phil}$ .

Syntax

There are two key parts of first order logic. The syntax determines which collections of symbols are legal expressions in first-order logic, while the semantics determine the meanings behind these expressions.

Alphabet

Unlike natural languages, such as English, the language of first-order logic is completely formal, so that it can be mechanically determined whether a given expression is legal. There are two key types of legal expressions: terms, which intuitively represent objects, and formulas, which intuitively express predicates that can be true or false. The terms and formulas of first-order logic are strings of symbols which together form the alphabet of the language. As with all formal languages, the nature of the symbols themselves is outside the scope of formal logic; they are often regarded simply as letters and punctuation symbols.

It is common to divide the symbols of the alphabet into logical symbols, which always have the same meaning, and non-logical symbols, whose meaning varies by interpretation. For example, the logical symbol  $\wedge$  always represents "and"; it is never interpreted as "or". On the other hand, a non-logical predicate symbol such as  $\text{Phil}(x)$  could be interpreted to mean " $x$  is a philosopher", " $x$  is a man named Philip", or any other unary predicate, depending on the interpretation at hand.

Logical symbols

There are several logical symbols in the alphabet, which vary by author but usually include:

- The quantifier symbols  $\forall$  and  $\exists$
- The logical connectives:  $\wedge$  for conjunction,  $\vee$  for disjunction,  $\rightarrow$  for implication,  $\leftrightarrow$  for biconditional,  $\neg$  for negation. Occasionally other logical connective symbols are included. Some authors use  $\Rightarrow$ , or  $Cpq$ , instead of  $\rightarrow$ , and  $\Leftrightarrow$ , or  $Epq$ , instead of  $\leftrightarrow$ , especially in contexts where  $\rightarrow$  is used for other purposes. Moreover, the horseshoe  $\supset$  may replace  $\rightarrow$ ; the triple-bar  $\equiv$  may replace  $\leftrightarrow$ , and a tilde ( $\sim$ ),  $Np$ , or  $Fpq$ , may replace  $\neg$ ;  $\parallel$ , or  $Apq$  may replace  $\vee$ ; and  $\&$ , or  $Kpq$ , may replace  $\wedge$ , especially if these symbols are not available for technical reasons.

Parentheses, brackets, and other punctuation symbols. The choice of such symbols varies depending on context.

An infinite set of variables, often denoted by lowercase letters at the end of the alphabet  $x, y, z, \dotsc$ . Subscripts are often used to distinguish variables:  $x_0, x_1, x_2, \dotsc$ .



An equality symbol (sometimes, identity symbol) =; see the section on equality below.

It should be noted that not all of these symbols are required - only one of the quantifiers, negation and conjunction, variables, brackets and equality suffice. There are numerous minor variations that may define additional logical symbols:

Sometimes the truth constants T,  $\forall pq$ , or  $\top$ , for "true" and F,  $\exists pq$ , or  $\perp$ , for "false" are included. Without any such logical operators of valence 0, these two constants can only be expressed using quantifiers.

### **Representing Simple Facts in logic:-**

In this approach, every non-logical symbol is of one of the following types.

1. A predicate symbol (or relation symbol) with some valence (or arity, number of arguments) greater than or equal to 0. These which are often denoted by uppercase letters  $P, Q, R, \dots$

- Relations of valence 0 can be identified with propositional variables. For example,  $P$ , which can stand for any statement?
- For example,  $P(x)$  is a predicate variable of valence 1. One possible interpretation is "x is a man".
- $Q(x,y)$  is a predicate variable of valence 2. Possible interpretations include "x is greater than y" and "x is the father of y".

2. A function symbol, with some valence greater than or equal to 0. These are often denoted by lowercase letters  $f, g, h, \dots$

- Examples:  $f(x)$  may be interpreted as for "the father of x". In arithmetic, it may stand for "-x". In set theory, it may stand for "the power set of x". In arithmetic,  $g(x,y)$  may stand for " $x+y$ ". In set theory, it may stand for "the union of x and y".
- Function symbols of valence 0 are called constant symbols, and are often denoted by lowercase letters at the beginning of the alphabet  $a, b, c, \dots$ . The symbol  $a$  may stand for Socrates. In arithmetic, it may stand for 0. In set theory, such a constant may stand for the empty set.

The traditional approach can be recovered in the modern approach by simply specifying the "custom" signature to consist of the traditional sequences of non-logical symbols.

#### Formation rules

The formation rules define the terms and formulas of first order logic. When terms and formulas are represented as strings of symbols, these rules can be used to write a formal grammar for terms and formulas. These rules are generally context-free (each production has a single symbol on the left side), except that the set of symbols may be allowed to be infinite and there may be many start symbols, for example the variables in the case of terms.

#### Terms

The set of terms is inductively defined by the following rules:

1. Variables. Any variable is a term.
2. Functions. Any expression  $f(t_1, \dots, t_n)$  of  $n$  arguments (where each argument  $t_i$  is a term and  $f$  is a function symbol of valence  $n$ ) is a term. In particular, symbols denoting individual constants are 0-ary function symbols, and are thus terms.

Only expressions which can be obtained by finitely many applications of rules 1 and 2 are terms. For example, no expression involving a predicate symbol is a term.

#### Formulas

The set of formulas (also called well-formed formulas or wffs) is inductively defined by the following rules:



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

1. Predicate symbols. If  $P$  is an  $n$ -ary predicate symbol and  $t_1, \dots, t_n$  are terms then  $P(t_1, \dots, t_n)$  is a formula.
2. Equality. If the equality symbol is considered part of logic, and  $t_1$  and  $t_2$  are terms, then  $t_1 = t_2$  is a formula.
3. Negation. If  $\phi$  is a formula, then  $\neg \phi$  is a formula.
4. Binary connectives. If  $\phi$  and  $\psi$  are formulas, then  $(\phi \rightarrow \psi)$  is a formula. Similar rules apply to other binary logical connectives.
5. Quantifiers. If  $\phi$  is a formula and  $x$  is a variable, then  $\forall x \phi$  and  $\exists x \phi$  are formulas.  
Only expressions which can be obtained by finitely many applications of rules 1-5 are formulas. The formulas obtained from the first two rules are said to be atomic formulas.

For example,

$$\forall x \forall y (P(f(x)) \rightarrow \neg (P(x) \rightarrow Q(f(y), x, z)))$$

is a formula, if  $f$  is a unary function symbol,  $P$  a unary predicate symbol, and  $Q$  a ternary predicate symbol. On the other hand,  $\forall x x \rightarrow$  is not a formula, although it is a string of symbols from the alphabet.

The role of the parentheses in the definition is to ensure that any formula can only be obtained in one way by following the inductive definition (in other words, there is a unique parse tree for each formula). This property is known as unique readability of formulas. There are many conventions for where parentheses are used in formulas. For example, some authors use colons or full stops instead of parentheses, or change the places in which parentheses are inserted. Each author's particular definition must be accompanied by a proof of unique readability.

This definition of a formula does not support defining an if-then-else function  $\text{ite}(c,a,b)$  where "c" is a condition expressed as a formula, that would return "a" if c is true, and "b" if it is false. This is because both predicates and functions can only accept terms as parameters, but the first parameter is a formula. Some languages built on first-order logic, such as SMT-LIB 2.0, add this.

Notational conventions

For convenience, conventions have been developed about the precedence of the logical operators, to avoid the need to write parentheses in some cases. These rules are similar to the order of operations in arithmetic. A common convention is:

- $\neg$  is evaluated first
- $\wedge$  and  $\vee$  are evaluated next
- Quantifiers are evaluated next
- $\rightarrow$  is evaluated last.

Moreover, extra punctuation not required by the definition may be inserted to make formulas easier to read. Thus the formula

$$(\neg \forall x P(x) \rightarrow \exists x \neg P(x))$$

might be written as

$$(\neg [\forall x P(x)]) \rightarrow \exists x [\neg P(x)].$$

In some fields, it is common to use infix notation for binary relations and functions, instead of the prefix notation defined above. For example, in arithmetic, one typically writes " $2 + 2 = 4$ " instead of " $=(+ (2,2), 4)$ ". It is common to regard formulas in infix notation as abbreviations for the corresponding formulas in prefix notation.

The definitions above use infix notation for binary connectives such as  $\rightarrow$ . A less common convention is Polish notation, in which one writes  $\rightarrow$ ,  $\wedge$ , and so on in front of their arguments rather than between them. This convention allows all punctuation symbols to be discarded. Polish notation is compact and elegant, but rarely used in practice because it is hard for humans to read it. In Polish notation, the formula



$$\forall x \forall y (P(f(x)) \rightarrow \neg(P(x) \rightarrow Q(f(y), x, z)))$$

becomes " $\forall x \forall y \neg (P(x) \rightarrow Q(f(y), x, z))$ ".

Example

In mathematics the language of ordered abelian groups has one constant symbol 0, one unary function symbol  $-$ , one binary function symbol  $+$ , and one binary relation symbol  $\leq$ . Then:

- The expressions  $+(x, y)$  and  $+(x, +(y, -z))$  are terms. These are usually written as  $x + y$  and  $x + y - z$ .
- The expressions  $+(x, y) = 0$  and  $\leq(x, +(y, -z)), +(x, y)$  are atomic formulas.

These are usually written as  $x + y = 0$  and  $x + y - z \leq x + y$ .

- The expression  $(\forall x \forall y \leq(x, y), z) \rightarrow \forall x \forall y +(x, y) = 0)$  is a formula, which is usually written as  $\forall x \forall y (x + y \leq z) \rightarrow \forall x \forall y (x + y = 0)$ .

Free and bound variables

In a formula, a variable may occur free or bound. Intuitively, a variable is free in a formula if it is not quantified: in  $\forall y P(x, y)$ , variable  $x$  is free while  $y$  is bound. The free and bound variables of a formula are defined inductively as follows.

1. Atomic formulas. If  $\phi$  is an atomic formula then  $x$  is free in  $\phi$  if and only if  $x$  occurs in  $\phi$ . Moreover, there are no bound variables in any atomic formula.
2. Negation.  $x$  is free in  $\neg \phi$  if and only if  $x$  is free in  $\phi$ .  $x$  is bound in  $\neg \phi$  if and only if  $x$  is bound in  $\phi$ .
3. Binary connectives.  $x$  is free in  $(\phi \rightarrow \psi)$  if and only if  $x$  is free in either  $\phi$  or  $\psi$ .  $x$  is bound in  $(\phi \rightarrow \psi)$  if and only if  $x$  is bound in either  $\phi$  or  $\psi$ . The same rule applies to any other binary connective in place of  $\rightarrow$ .
4. Quantifiers.  $x$  is free in  $\forall y \phi$  if and only if  $x$  is free in  $\phi$  and  $x$  is a different symbol from  $y$ . Also,  $x$  is bound in  $\forall y \phi$  if and only if  $x$  is  $y$  or  $x$  is bound in  $\phi$ . The same rule holds with  $\exists$  in place of  $\forall$ .

For example, in  $\forall x \forall y (P(x) \rightarrow Q(x, f(x), z))$ ,  $x$  and  $y$  are bound variables,  $z$  is a free variable, and  $w$  is neither because it does not occur in the formula.

Freeness and boundness can be also specialized to specific occurrences of variables in a formula. For example, in  $P(x) \rightarrow \forall x Q(x)$ , the first occurrence of  $x$  is free while the second is bound. In other words, the  $x$  in  $P(x)$  is free while the  $x$  in  $\forall x Q(x)$  is bound.

A formula in first-order logic with no free variables is called a first-order sentence. These are the formulas that will have well-defined truth values under an interpretation. For example, whether a formula such as  $\text{Phil}(x)$  is true must depend on what  $x$  represents. But the sentence  $\exists x \text{Phil}(x)$  will be either true or false in a given interpretation.

### First-order logic

First-order logic is symbolized reasoning in which each sentence, or statement, is broken down into a subject and a predicate. The predicate modifies or defines the properties of the subject. In first-order logic, a predicate can only refer to a single subject. First-order logic is also known as first-order predicate calculus or first-order functional calculus.

A sentence in first-order logic is written in the form  $Px$  or  $P(x)$ , where  $P$  is the predicate and  $x$  is the subject, represented as a variable. Complete sentences are logically combined and manipulated according to the same rules as those used in Boolean algebra.

In first-order logic, a sentence can be structured using the universal quantifier (symbolized  $\forall$ ) or the existential quantifier ( $\exists$ ). Consider a subject that is a variable represented by  $x$ . Let  $A$  be a predicate "is an apple,"  $F$  be a predicate "is a fruit,"  $S$  be a predicate "is sour", and  $M$  be a predicate "is mushy." Then we can say





तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

$$\forall x : Ax \implies Fx$$

which translates to "For all x, if x is an apple, then x is a fruit." We can also say such things as

$$\exists x : Fx \implies Ax$$

$$\exists x : Ax \implies Sx$$

$$\exists x : Ax \implies Mx$$

where the existential quantifier translates as "For some."

First-order logic can be useful in the creation of computer programs. It is also of interest to researchers in artificial intelligence (AI). There are more powerful forms of logic, but first-order logic is adequate for most everyday reasoning. The Incompleteness Theorem, proven in 1930, demonstrates that first-order logic is in general undecidable. That means there exist statements in this logic form that, under certain conditions, cannot be proven either true or false.

### Representing instances and isa relationship:-

#### THE "ISA" RELATIONSHIP

After establishing a class hierarchy with the Entity/Relation model, the principle of generalization is used to identify the class hierarchy and the level of abstraction associated with each class. Generalization implies a successive refinement of the class, allowing the super classes of objects to inherit the data attributes and behaviors which apply to the lower levels of the class. Generalization establishes "taxonomy hierarchies", which organize the classes according to their characteristics, usually in increasing levels of detail. Generalization begins at a very general level, and proceeds to a specific level, with each sub-level having its own unique data attributes and behaviors.

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.

### Computable function and predicate:-

**Computable functions** are the basic objects of study in computability theory. Computable functions are the formalized analogue of the intuitive notion of algorithm. They are used to discuss computability without referring to any concrete model of computation such as Turing machines or register machines. Any definition, however, must make reference to some specific model of computation but all valid definitions yield the same class of functions. Particular models of computability that give rise to the set of computable functions are the Turing-computable functions and the  $\lambda$ -recursive functions.

Before the precise definition of computable function, mathematicians often used the informal term *effectively calculable*. This term has since come to be identified with the computable functions. Note that the effective computability of these functions does not imply that they can be *efficiently* computed (i.e. computed within a reasonable amount of time). In fact, for some effectively calculable functions it can be shown that any algorithm that computes them will be very inefficient in the sense that the running time of the algorithm increases exponentially with the length of the input. The fields of feasible computability and computational complexity study functions that can be computed efficiently.

According to the Church-Turing thesis, computable functions are exactly the functions that can be calculated using a mechanical calculation device given unlimited amounts of time and storage space. Equivalently, this thesis states that any function which has an algorithm is computable. Note that an



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
**Institute of Management & Technology**  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

algorithm in this sense is understood to be a sequence of steps a person with unlimited time and an infinite supply of pen and paper could follow.

### UNIT - III

#### **Natural language processing: Introduction syntactic processing:-**

##### **Definitions: Grammar**

A grammar of a language is a scheme for specifying the sentences in that language. It indicates the syntactic rules for combining words into well-formed phrases and clauses. The theory of *generative grammar* [Chomsky, 1957] had a profound effect on linguistic research, including AI work in computational linguistics.

##### **Parsing**

Parsing is the "de-linearization" of linguistic input; that is, the use of grammatical rules and other knowledge sources to determine the functions of words in the input sentence. Usually a parser produces a data structure like a *derivation tree* to represent the structural meaning of a sentence.

##### **definite clause grammars**

Grammars in which rules are written as PROLOG clauses and the PROLOG interpreter is used to perform top-down, depth-first parsing.

##### **ATN**

[Woods, 1970] An *augmented transition network* is one in which parsing is described as the transition from a start state to a final state in a transition network corresponding to an English grammar. ATN was first used in LUNAR. An ATN is similar to a FSM in which the class of labels that can be attached to transition arcs has been augmented. Arcs in an ATN may contain words, categories (e.g., *noun*), they may push to other networks, or perform procedures. ATNs may contain registers.

ATN grammars can generate all recursively enumerable sets.

One problem with ATNs (especially in speech recognition) is that their procedurality limits their effectiveness. It is sometimes better to parse a sentence based on clearly recognized parts when earlier parts are still unknown.

##### **RTN**

*Recursive transition networks* generalize finite state automata by allowing nondeterministic transitions between two states to be taken via a recursive jump to a start state. RTNs recognize exactly the class of CFLs. ATNs are RTNs that also have a finite set of registers and actions that can set registers to input words, corresponding lexical entries, or to some function of the content of other registers. Recursive calls to the network can pass values back to the calling level.

##### **unification grammar**

A declarative approach to representing grammars. The parser does *matching* of sentence constituents to grammar rules, and builds structures corresponding to the result of combining constituents. A unification operator can be applied to the graphs, which is similar to logical unification. Two graphs unify if, recursively, all of their subgraphs unify.

##### **Montague semantics**

(Also, *compositional semantics*) For every step in the syntactic parsing process, there is a corresponding step in semantic interpretation. Each time syntactic constituents are combined to



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

form a larger syntactic unit, their corresponding semantic interpretation can be combined to form a larger semantic unit. The clearest application of this idea is the work of Montague.

The compositional approach to defining semantic interpretation has proved to be a very powerful idea. Unfortunately, there are some linguistic constructions (such as quantification) that cannot be accounted for in this way.

#### Phoneme

A smallest distinguishable unit of speech.

#### Morpheme

A meaningful linguistic unit that contains no smaller meaningful parts.

#### Speech recognition

Recognizing speech without domain knowledge.

#### Speech understanding

Recognizing speech with domain knowledge.

#### Illocutionary force

A force present in an utterance in which the speaker's intention is distinct from what is actually said. A *promise* has illocutionary force -- it has the propositional content that I will do a certain thing at some particular time, but the utterance is actually performing a speech act that neither describes a state of the world nor makes a prediction about the state of the world.

#### Hermeneutic connection

*Hermeneutics* is the study of interpretation. One of the fundamental insights of hermeneutics is the importance of *pre-understanding*, to wit, that understanding rises and evolves through acts of interpretation. This circle, in which understanding is necessary for interpretation, which in turn creates understanding, is called the *hermeneutic circle*.

#### Garden path sentences

Sentences that require conscious backtracking. Examples: *The horse raced past the barn fell*, *The cake baked by the baker burnt*, and *Have the students who failed the exam take the supplemental*.

#### Anaphora

The use of a word to refer to previously-mentioned entities, e.g., *The boys and I went over to Frank's, because they needed to talk to him*.

#### Prosody

The particular fluctuations in stress and intonation in a sentence.

#### Overview

To *understand* something is to transform it from one representation into another, where this latter representation is chosen to correspond to a set of available actions that could be performed, and for which a mapping is designed so that for each event an *appropriate* action will be performed.

#### Advantages of Natural Language Interfaces

Natural language is only one medium for human-machine interaction, but has several obvious and desirable properties:

1. It provides an immediate vocabulary for talking about the contents of the computer.
2. It provides a means of accessing information in the computer independently of its structure and encodings.
3. It shields the user from the formal access language of the underlying system.



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

4. It is available with a minimum of training.

#### The Hardness of Natural Language

There are several major reasons why natural language understanding is a difficult problem. They include:

1. The complexity of the target representation into which the matching is being done. Extracting meaningful information often requires the use of additional knowledge.
2. The type of mapping: one-to-one, many-to-one, one-to-many, or many-to-many. One-to-many mappings require a great deal of domain knowledge beyond the input to make the correct choice among target representations. So for example, the word *tall* in the phrase "a tall giraffe" has a different meaning than in "a tall poodle." English requires many-to-many mappings.
3. The level of interaction of the components of the source representation. In many natural language sentences, changing a single word can alter the interpretation of the entire structure. As the number of interactions increases, so does the complexity of the mapping.
4. The presence of noise in the input to the understander. We rarely listen to one another against a silent background. Thus *speech recognition* is a necessary precursor to speech understanding.
5. The modifier attachment problem. (This arises because sentences aren't inherently hierarchical, I'd say -- POD.) The sentence *Give me all the employees in a division making more than \$50,000* doesn't make it clear whether the speaker wants all employees making more than \$50,000, or only those in divisions making more than \$50,000.
6. The quantifier scoping problem. Words such as "the," "each," or "what" can have several readings.
7. Elliptical utterances. The interpretation of a query may depend on previous queries and their interpretations. E.g., asking *Who is the manager of the automobile division* and then saying, *of aircraft?*

#### Speech Acts

A view of language in which utterances can be understood as acts rather than representations. In giving a command or making a promise, for example, a speaker is entering into an interaction pattern, playing a certain role, committing both the speaker and the hearer to future actions.

The implications of this view include:

1. It allows us to deal with the consequences of actually making statements.
2. It shifts reasoning away from the objective/subjective dichotomy (into the domain of interaction as opposed to objective truth).
3. It places a central emphasis on the potential for further articulation of unstated background. Human interaction *always* takes place in an unarticulated background; the background can never be made fully explicit.

We often use *indirect speech acts* (e.g., *Do you know what time it is?* or *It's cold in here!*)

Searle presents a linguistic theory of indirect speech acts that includes certain *conversational postulates* about conversation that are shared by all speakers. These include

- Sincerity conditions For *A* to request *B* to do *R*, *A* must want *B* to do *R*, *A* must assume *B* can do *R*, and that *B* is willing to do *R*. We can also assume that *B* would not have done *R* anyway.
- Reasonableness conditions *A* must have reasons for assuming all of the above. Reasonableness conditions often provide the basis for challenging a request (e.g., *Why do you want me to open the door?*)
- Appropriateness conditions The statement must provide the correct amount of information and must be polite. If we ask who won the race and get something like *Someone with long, dark hair* we can assume the name wasn't known.

#### Natural Language Processing:-



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

Language processing can be divided into two tasks:

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

The steps in the process of natural language understanding are:

**Morphological analysis**

Individual words are analyzed into their components, and non-word tokens (such as punctuation) are separated from the words. For example, in the phrase "Bill's house" the proper noun "Bill" is separated from the possessive suffix "'s."

**Syntactic analysis**

Linear sequences of words are transformed into structures that show how the words relate to one another. This parsing step converts the flat list of words of the sentence into a structure that defines the units represented by that list. Constraints imposed include word order ("manager the key" is an illegal constituent in the sentence "I gave the manager the key"); number agreement; case agreement.

**Semantic analysis**

The structures created by the syntactic analyzer are assigned meanings. In most universes, the sentence "Colorless green ideas sleep furiously" [Chomsky, 1957] would be rejected as *semantically anomalous*. This step must map individual words into appropriate objects in the knowledge base, and must create the correct structures to correspond to the way the meanings of the individual words combine with each other.

**Discourse integration**

The meaning of an individual sentence may depend on the sentences that precede it and may influence the sentences yet to come. The entities involved in the sentence must either have been introduced explicitly or they must be related to entities that were. The overall discourse must be coherent.

**Pragmatic analysis**

The structure representing what was said is reinterpreted to determine what was actually meant.

**Signal Processing**

Signal processing is the task of taking a spoken bit of language and turning it into a sequence of words. The problems involved digitizing the signal, and then distinguishing word segments that can be assembled into whole words.

The linguistic elements handled by signal processing are *phonemes*. Identifying these sounds in real-time is relatively easy, since there isn't much data in auditory signals. However, assembling them into words can be difficult, e.g.:

It's very hard to wreck a nice beach.

### **Syntactic Processing:-**

Syntactic parsing determines the structure of the sentence being analyzed. Syntactic analysis involves parsing the sentence to extract whatever information the word order contains. Syntactic parsing is computationally less expensive than semantic processing.

A grammar is a declarative representation that defines the syntactic facts of a language. The most common way to represent grammars is as a set of production rules, and the simplest structure for them to build is a *parse tree* which records the rules and how they are matched.



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

Sometimes backtracking is required (e.g., *The horse raced past the barn fell*), and sometimes multiple interpretations may exist for the beginning of a sentence (e.g., *Have the students who missed the exam --* ).

*Example:* Syntactic processing interprets the difference between "John hit Mary" and "Mary hit John."  
Semantic Analysis

After (or sometimes in conjunction with) syntactic processing, we must still produce a representation of the *meaning* of a sentence, based upon the meanings of the words in it. The following steps are usually taken to do this:

Lexical processing

Look up the individual words in a dictionary. It may not be possible to choose a single correct meaning, since there may be more than one. The process of determining the correct meaning of individual words is called *word sense disambiguation* or *lexical disambiguation*. For example, "I'll meet you at the diamond" can be understood since *at* requires either a time or a location. This usually leads to *preference semantics* when it is not clear which definition we should prefer.

Sentence-level processing

There are several approaches to sentence-level processing. These include semantic grammars, case grammars, and conceptual dependencies.

*Example:* Semantic processing determines the differences between such sentences as "The pig is in the pen" and "The ink is in the pen."

Discourse and Pragmatic Processing

To understand most sentences, it is necessary to know the discourse and pragmatic context in which it was uttered. In general, for a program to participate intelligently in a dialog, it must be able to represent its own beliefs about the world, as well as the beliefs of others (and their beliefs about its beliefs, and so on).

The context of *goals* and *plans* can be used to aid understanding. Plan recognition has served as the basis for many understanding programs -- PAM is an early example.

Speech acts can be axiomatized just as other operators in written language, except that they require modal operators to describe states of belief, knowledge, et al.

Grammars

Chomsky delineated four types of grammars, numbered 0 to 3.

- Type 0 Most general, with no restrictions on the form that rewrite rules can take. Languages generated by type 0 grammars are exactly those recognized by Turing machines.
- Type 1 Context-sensitive. The right-hand side of the production must contain at least as many symbols as the left-hand side.
- Type 2 Context-free. The left-hand side of a production must contain exactly one symbol, a nonterminal.
- Type 3 Regular expressions. All productions are of the form  $X \rightarrow aY$  or  $X \rightarrow a$ . Exactly those languages recognized by finite state automata.

Transformational Grammars

Transformational grammars were introduced by Chomsky in 1957 in his article on syntactic structures. Utterance is characterized as the surface manifestation of a "deeper" structure representing "meaning" of the sentence. The deep structure can undergo a variety of *transformations* of form (e.g., word order, endings, etc.) on its way up, while retaining its essential meaning.

Transformational grammars have three components. The first is a phrase-structure grammar generating strings of morphemes representing simple, declarative, active sentences, each with an associated phrase marker or derivation tree. The second is a set of *transformational rules* rearranging these strings and



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

adding or deleting morphemes to form representations of the full variety of legal sentences. Finally, a sequence of morphophonemic rules would map each sentence representation to a string of phonemes. (The first level strikes me as highly similar to Schank's conceptual dependency representations -- POD)

A simple example:

The + boy + TENSE + eat (derived by grammar)

The + boy + eat + PAST (transformation rule)

The boy ate. (morphophonemic rule)

### **Semantic processing:-**

Semantic grammars combine syntactic, semantic, and pragmatic knowledge into a single set of rules in the form of a grammar. This is usually just a context-free grammar in which the choice of nonterminal and production rules is governed by semantic as well as syntactic function. This is usually good for restricted natural-language interfaces.

### **Systemic Grammars**

Systemic grammars distinguish three general functions of language, all of which are ordinarily served by every act of speech.

The *ideational function* serves for the expression of content. It says something about the speaker's experience of the world. Analyzing a clause in terms of its ideational function involves asking questions like: What kind of process does the clause describe -- an action, a mental process, or a relation? Who is the actor? Are there other participants in the process?

The *interpersonal function* relates to the purpose of the utterance. The speaker may be asking a question, answering one, making a request, giving information, or expressing an opinion. The *mood system* of English grammar expresses these possibilities in terms of categories such as statement, question, command, and exclamation.

Lastly, the *textual function* reflects the need for coherence in language use. Concepts for use in textual terms include theme (the element the speaker chooses to put at the beginning of a clause) and change (distinction between what is new in a message and what is assumed to be given).

### **Case Grammars**

[Fillmore, 1968] Fillmore characterized the relationships between the verb and noun phrase as "semantically relevant syntactic relationships" and called them *cases*. The case assignment comes from the deep structure, even though the surface structure is different. (So, for example, the sentence "John opened the door with the key" is semantically equivalent to "The door was opened by John with the key.") The structure that is built by the parse contains some semantic information, although further interpretation may be necessary. Grammar rules are written to describe syntactic rather than semantic regularities. But the structures the rules produce correspond to semantic relations rather than to strictly syntactic ones. For example:

S -> NP VP

Parsing using case grammars is usually *expectation-driven*. Once the verb of the sentence has been located, it can be used to predict the noun phrases that will occur and to determine the relationship of those phrases to the rest of the sentence.

### **Unification Grammars**

A wide range of recent approaches to natural language processing can be described in terms of unification grammars. Most such systems share a common basis -- context-free grammars -- and are extended using the concept of *unification* (or matching) on partially-specified tree structures.

The principal problem with earlier augmented parsing systems (such as ATNs) is that they force certain search strategies; ATNs usually take a top-down approach. This works poorly for other approaches; for



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

example, island-driven search centering on words recognized with high confidence. If the search isn't sequential, many of the registers in an ATN will not be set appropriately and arcs can't be followed as needed.

Restrictions can be placed on augmented systems to remedy this problem. In particular, if registers are not allowed to be reassigned once they have a value, many problems disappear because a system can then be developed that delays the evaluation of tests until all the necessary registers are set. Instead of treating register assignments like variable assignments in a programming language, unification-based systems use the notion of unification between logical variables.

The most notable advantages of this approach stem from the independence of the grammar from a particular parsing algorithm, so each rule in the grammar can be better isolated as a unit of study and modification.

A simple unification system can be described as follows. It is an extension of a standard CFG in which each rule is annotated with a set of unification equations. Specifically, two register-value structures unify if each of the specified register values is consistent with the same register's value in the other structure. Registers specified in one and not in the other are simply copied.

Example:

(S MAIN-V (VERB ROOT LOVE)) and (S MAIN-V (VERB FORM en) NUM {3s})

unify to produce the structure

(S MAIN-V (VERB ROOT LOVE FORM en) NUM {3s})

All unification equations are of the form *structure* = *structure*. The structure can be defined precisely by defining each slot name as a function from its containing constituent to its value; this can be represented with directed, acyclic graphs in which each constituent and value is a node, and the slots are labeled arcs.

Parsing

Parsing Systems

Template matching

Parsing by matching the input against a series of predefined templates. Used in programs like SIR, STUDENT, ELIZA.

Phrase-structure grammar parsers

Usually used context-free grammars with various combinations of the parsing techniques mentioned.

Transformational grammar parsers

More comprehensive than phrase-structure grammars, but failed because of combinatorial explosion.

Extended grammar parsers

Extended the concept of phrase-structure rules and derivations by adding mechanisms for more complex representations and manipulation of sentences. This includes ATNs and charts.

Semantic grammar parsers

Modification to the traditional phrase structure approach that uses concepts like *noun*, *verb*, etc., into a semantic grammar for a particular domain, using concepts like *destination*, *flight*, *flight-time*. Used by Lifer and Sophie.

Grammar-less parsers

Ad hoc systems that use special procedures (e.g., SHRDLU).

Augmented Transition Networks

As described above, ATNs are RTNs that allow tests on their arcs, and have registers to record input values or functions of register values during processing. The basic bottom-up ATN parsing algorithm is as follows.





तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

1. Set the ATN pointer to the start state and the sentence pointer to the beginning of the sentence being parsed. The current node is always the first element of the ATN pointer.
2. Select an arc out of the current node. In order to legally traverse the arc, it must satisfy the following conditions:
  1. Any associated test must be satisfied by the current variable values.
  2. If the arc is labeled with a word category (e.g., noun) the current word must be a member of that category.
3. Execute the actions associated with the arc. In addition, do the following, based on the type of arc:
  1. If the arc is a word category, update the current position in the sentence by one word and change the current node to the destination of the arc.
  2. If the arc corresponds to another ATN, push the starting node of that ATN onto the ATN pointer.
  3. If the arc is jump, change the current node to the destination of the arc.
  4. If the arc is done, pop the current node off of the ATN pointer and set \* to the value returned by this node. If the ATN pointer is now empty and all of the text has been processed, return \*. If the ATN pointer is empty and text remains, fail. Otherwise, return to step 2.

#### Conceptual parsing (CD)

[Schank] Schank developed conceptual dependency (CD) as a representation for the meanings of phrases and sentences. The basic tenet of CD theory is

For any two sentences that are identical in meaning, regardless of language, there should be only one representation of that meaning in CD.

Schank thus accepts the early machine translation concept of an *interlingua* or intermediate language. The other, more dubious (POD) idea in CD theory is

Any information in a sentence that is implicit must be made explicit in the representation of the meaning of the sentence.

Primitive acts in conceptual dependencies include: PTRANS (physical transfer), PROPEL (physical force), MTRANS (mental transfers, like *tell*), etc. There are 11 of these primitive acts [1977]. Relations among concepts are called *dependencies*, of which there are a fixed number (e.g., "R" for recipient-donor dependency.)

Here is the translation of the sentence *John gives Mary a book*:

```
o      R +---> Mary
John <====> ATRANS <--- book <---|
      +---> John
```

In the CD, "o" means that the book is the object of the ATRANS, and the three-pointed "R" relation is a recipient-donor dependency between Mary, John, and the book.

Conceptual dependencies also model *primitive states*, which are integer-valued variables such as HEALTH or MENTAL STATE. States and actions can be combined, so for example *John told Mary that Bill was happy* can be represented as

John MTRANS (Bill BE MENTAL-STATE(5)) to Mary.

In CDs, syntactic and semantic knowledge are combined into a single interpretation system that is driven by semantic knowledge. Syntactic parsing is subordinated to semantic interpretation, which is usually used to set up strong expectations for particular sentence structures. Parsing is heavily driven by a set of expectations that are set up on the basis of the sentence's main verb. Because the representation of a verb in CD is at a lower level than that of a verb in a case grammar, CD usually provides a greater degree of



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

predictive

power.

### Machine Translation

Simply using dictionaries doesn't solve the problem. The other troubles are

1. Many words have several translations depending on context, and some don't have any (e.g., Japanese has two different words, "hot water" and "cold water").
2. Word orders vary from language to language.
3. Idiomatic expressions cannot be translated word-for-word but must be translated *in toto*.

A proposed approach for translating between arbitrary languages was to go through an intermediate "universal language" or *interlingua*. However, without commonsense world knowledge, certain kinds of translations are impossible. Bar-Hillel gives the example of "The pen is in the box" as opposed to "The box is in the pen." No translator without world understanding could parse these sentences correctly.

### Speech Understanding

Since speech is our most natural form of communication, using spoken language to access computers has long been an important research goal. Originally, in the 1960's, research was done on *isolated word recognition* before the later development of *speech understanding* systems.

A physical problem with such systems is noise. Microphone and background noise can cause interference with the recording of the spoken utterance. Another problem is that the speaker doesn't always pronounce the same word in the same way. Sometimes whole syllables are dropped or "swallowed" at word boundaries. In fact, finding the boundaries between words in a connected-speech signal is one of the difficult problems in speech understanding.

With increased information available to the hearer beyond the acoustic signal (such as context), expectations about the content can be formed.

The ARPA Speech Understanding Research Program

In the early 1970's, the Advanced Research Projects Agency of the U.S. Department of Defense funded a five-year program in speech understanding research (the *SUR project*). The systems to be designed were to accept normally spoken sentences in a constrained domain with a 1,000 word vocabulary, and were to respond in reasonable time with less than 10% error. (This was one of the few times that AI programs had any design objectives specified before development.)

Natural language processing (NLP) is a field of computer science and linguistics concerned with the interactions between computers and human (natural) languages; Specifically, the process of a computer extracting meaningful information from natural language input and/or producing natural language output. It began as a branch of artificial intelligence. In theory, natural language processing is a very attractive method of human-computer interaction. Natural language understanding is sometimes referred to as an AI-complete problem because it seems to require extensive knowledge about the outside world and the ability to manipulate it.

Whether NLP is distinct from, or identical to, the field of computational linguistics is a matter of perspective. The Association for Computational Linguistics defines the latter as focusing on the theoretical aspects of NLP. On the other hand, the open-access journal "Computational Linguistics", styles itself as "the longest running publication devoted exclusively to the design and analysis of natural language processing systems" (Computational Linguistics (Journal))

Modern NLP algorithms are grounded in machine learning, especially statistical machine learning. Research into modern statistical NLP algorithms requires an understanding of a number of disparate



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

fields, including linguistics, computer science, and statistics. For a discussion of the types of algorithms currently used in NLP, see the article on pattern recognition.

semantic analysis (LSA) is a technique in natural language processing, in particular in vectorial semantics, of analyzing relationships between a set of documents and the terms they contain by producing a set of concepts related to the documents and terms. LSA assumes that words that are close in meaning will occur close together in text. A matrix containing word counts per paragraph (rows represent unique words and columns represent each paragraph) is constructed from a large piece of text and a mathematical technique called singular value decomposition (SVD) is used to reduce the number of columns while preserving the similarity structure among rows. Words are then compared by taking the cosine of the angle between the two vectors formed by any two rows. Values close to 1 represent very similar words while values close to 0 represent very dissimilar words

### **Discourse and pragmatic processing:-**

Discourse and Pragmatics

ÉTo actually understand the sentence, we must consider the context in which it was uttered/written.

ÉTo help in this, there are a number of important relationships that may hold between phrases and part of their discourse context.

November 06 Discourse & Pragmatic Processing 2 Discourse and Pragmatics

ÉIdentical entities.

ó Bill had a red balloon. John wanted it.

ÉParts of entities.

ó Sue opened the book she had just bought. The title page was torn.

ÉParts of actions.

ó John went on a business trip. He left on an early morning flight.

November 06 Discourse & Pragmatic Processing 3 Discourse and Pragmatics

ÉEntities involved in actions.

ó My house was broken into last week. They took the tv and stereo.

ÉElements of sets.

ó The decals in stock are stars, the moon and a flag. I'dl take two moons.

ÉNames of individuals.

ó Dave went to the movies. November 06 Discourse & Pragmatic Processing 4 Discourse and Pragmatics

ÉCausal chains.

ó There was a big snow storm yesterday. The schools were closed today.

ÉPlanning sequences.

ó Sally wanted a new car. She decided to get a job.

ÉIllocutionary force.

ó It sure is cold in here. November 06 Discourse & Pragmatic Processing 5 Discourse and Pragmatics

ÉTo recognize these, a lot of knowledge of the world being discussed is required.

ÉLarge knowledge bases are required for systems that understand multiple sentences.

ó Organization is critical.

ÉThe types of knowledge being dealt with include:-

November 06 Discourse & Pragmatic Processing 6 Discourse and Pragmatics

ÉThe current focus of the dialog.

ÉA model of each participant's current beliefs.

ÉThe goal-driven character of dialogue.

ÉThe rules of conversation shared by all participants.

ÉInclude reasoning about objects, events, goals, plans, intentions, beliefs and likelihoods



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

ó Seen some of these before.

### **Learning: Introduction learning:-**

Machine learning, a branch of artificial intelligence, is a scientific discipline concerned with the design and development of algorithms that allow computers to evolve behaviors based on empirical data, such as from sensor data or databases. A learner can take advantage of examples (data) to capture characteristics of interest of their unknown underlying probability distribution. Data can be seen as examples that illustrate relations between observed variables. A major focus of machine learning research is to automatically learn to recognize complex patterns and make intelligent decisions based on data; the difficulty lies in the fact that the set of all possible behaviors given all possible inputs is too large to be covered by the set of observed examples (training data). Hence the learner must generalize from the given examples, so as to be able to produce a useful output in new cases.

### **Rote learning:-**

Rote learning is a learning technique which focuses on memorization without the use of meaning as a basis to store information. The major practice involved in rote learning is learning by repetition by which students commit information to memory in a highly structured way. The idea is that one will be able to quickly recall the meaning of the material the more one repeats it. Some of the alternatives to rote learning include meaningful learning, associative learning, and active learning.

Rote learning vs. critical thinking

Rote methods are routinely used when quick memorization is required, such as learning one's lines in a play or memorizing a telephone number. Rote learning is widely used in the mastery of foundational knowledge. Examples of school topics where rote learning is frequently used include phonics in reading, the periodic table in chemistry, multiplication tables in mathematics, anatomy in medicine, cases or statutes in law, basic formulae in any science, etc. By definition, rote learning eschews comprehension, so by itself it is an ineffective tool in mastering any complex subject at an advanced level. For instance, one illustration of Rote learning can be observed in preparing quickly for exams, a technique which may be colloquially referred to as "cramming".

Rote learning is sometimes disparaged with the derogative terms *parrot fashion*, *regurgitation*, *cramming*, or *mugging* because one who engages in rote learning may give the wrong impression of having understood what they have written or said. It is strongly discouraged by many new curriculum standards. For example, science and mathematics standards in the United States specifically emphasize the importance of deep understanding over the mere recall of facts, which is seen to be less important, although advocates of traditional education have criticized the new American standards as slighting learning basic facts and elementary arithmetic, and replacing content with process-based skills.

"When calculators can do multi digit long division in a microsecond, graph complicated functions at the push of a button, and instantaneously calculate derivatives and [[integrals, serious questions arise about what is important in the mathematics curriculum and what it means to learn mathematics. More than ever, mathematics must include the mastery of concepts instead of mere memorization and the following of procedures. More than ever, school mathematics must include an understanding of how to use technology to arrive meaningfully at solutions to problems instead of endless attention to increasingly outdated computational tedium."

Rote learning as a necessity

With some material rote learning is the only way to learn it in a short time; for example, when learning the Greek alphabet or lists of vocabulary words Similarly, when learning the conjugation of foreign irregular verbs, the morphology is often too subtle to be learned explicitly in a short time. However, as in



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

the alphabet example, learning where the alphabet came from may help one to grasp the concept of it and therefore memorize it. (Native speakers and speakers with a lot of experience usually get an intuitive grasp of those subtle rules and are able to conjugate even irregular verbs that they have never heard before.)

The source transmission could be auditory or visual, and is usually in the form of short bits such as rhyming phrases (but rhyming is not a prerequisite), rather than chunks of text large enough to make lengthy paragraphs. Brevity is not always the case with rote learning. For example, many Americans can recite their National Anthem, or even the much more lengthy Preamble to the United States Constitution. Their ability to do so can be attributed, at least in some part, to having been assimilated by rote learning. The repeated stimulus of hearing it recited in public, on TV, at a sporting event, etc. has caused the mere sound of the phrasing of the words and inflections to be "written", as if hammer-to-stone, into the long-term memory. Excessive repetition within a limited time frame can actually be counter-productive to learning, through an effect termed semantic satiation.

### **Learning by Taking Advice:-**

This is a simple form of learning. Suppose a programmer writes a set of instructions to instruct the computer what to do, the programmer is a teacher and the computer is a student. Once learned (i.e. programmed), the system will be in a position to do new things.

The advice may come from many sources: human experts, internet to name a few. This type of learning requires more inference than rote learning. The knowledge must be transformed into an operational form before stored in the knowledge base. Moreover the reliability of the source of knowledge should be considered.

The system should ensure that the new knowledge is conflicting with the existing knowledge. FOO (First Operational Operationaliser), for example, is a learning system which is used to learn the game of Hearts. It converts the advice which is in the form of principles, problems, and methods into effective executable (LISP) procedures (or knowledge). Now this knowledge is ready to use.

### **Learning :-**

What is Learning?

Learning is an important area in AI, perhaps more so than planning.

- Problems are hard -- harder than planning.
- Recognised Solutions are not as common as planning.
- A goal of AI is to enable computers that can be taught rather than programmed.

*Learning* is a an area of AI that focusses on processes of self-improvement.

Information processes that improve their performance or enlarge their knowledge bases are said to *learn*.

*Why is it hard?*

- Intelligence implies that an organism or machine must be able to adapt to new situations.
- It must be able to learn to do new things.
- This requires knowledge acquisition, inference, updating/refinement of knowledge base, acquisition of heuristics, applying faster searches, *etc.*

How can we learn?

Many approaches have been taken to attempt to provide a machine with learning capabilities. This is because learning tasks cover a wide range of phenomena.

Listed below are a few examples of how one may learn. We will look at these in detail shortly

Skill refinement

-- one can learn by practicing, *e.g playing the piano.*



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

Knowledge acquisition

-- one can learn by experience and by storing the experience in a knowledge base. One basic example of this type is rote learning.

Taking advice

-- Similar to rote learning although the knowledge that is input may need to be transformed (or *operationalised*) in order to be used effectively.

### Learning in problem solving :-

Problem Solving

-- if we solve a problem one may learn from this experience. The next time we see a similar problem we can solve it more efficiently. This does not usually involve gathering new knowledge but may involve reorganisation of data or remembering how to achieve to solution.

Induction

-- One can learn from *examples*. Humans often classify things in the world without knowing explicit rules. Usually involves a teacher or trainer to aid the classification.

Discovery

-- Here one learns knowledge without the aid of a teacher.

Analogy

-- If a system can recognise similarities in information already stored then it may be able to transfer some knowledge to improve to solution of the task in hand.

### Rote Learning:-

Rote Learning is basically *memorisation*.

- Saving knowledge so it can be used again.
- Retrieval is the only problem.
- No repeated computation, inference or query is necessary.

A simple example of rote learning is *caching*

- Store computed values (or large piece of data)
- Recall this information when required by computation.
- Significant time savings can be achieved.
- Many AI programs (as well as more general ones) have used caching very effectively.

Memorisation is a key necessity for learning:

- It is a basic necessity for any intelligent program -- is it a separate learning process?
- Memorisation can be a complex subject -- how best to store knowledge?

Samuel's Checkers program employed rote learning (it also used parameter adjustment which will be discussed shortly).

- A minimax search was used to explore the game tree.
- Time constraints do not permit complete searches.
- It *records* board positions and scores at search ends.
- Now if the same board position arises later in the game the stored value can be recalled and the end effect is that more deeper searched have occurred.

Rote learning is basically a simple process. However it does illustrate some issues that are relevant to more complex learning issues.

Organisation

-- access of the stored value must be faster than it would be to recompute it. Methods such as hashing, indexing and sorting can be employed to enable this.



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

E.g Samuel's program indexed board positions by noting the number of pieces.

#### Generalisation

-- The number of potentially stored objects can be very large. We may need to generalise some information to make the problem manageable.

E.g Samuel's program stored game positions only for white to move. Also rotations along diagonals are combined.

#### Stability of the Environment

-- Rote learning is not very effective in a rapidly changing environment. If the environment does change then we must detect and record exactly what has changed -- *the frame problem*.

#### Store v Compute

Rote Learning must not decrease the efficiency of the system.

We must be able to decide whether it is worth storing the value in the first place.

Consider the case of multiplication -- it is quicker to recompute the product of two numbers rather than store a large multiplication table.

How can we decide?

#### Cost-benefit analysis

-- Decide when the information is first available whether it should be stored. An analysis could weigh up amount of storage required, cost of computation, likelihood of recall.

#### Selective forgetting

-- here we allow the information to be stored initially and decide later if we retain it. Clearly the frequency of reuse is a good measure. We could tag an object with its *time of last use*. If the cache memory is full and we wish to add a new item we remove the least recently used object. Variations could include some form of cost-benefit analysis to decide if the object should be removed.

#### Learning by Taking Advice:-

The idea of advice taking in AI based learning was proposed as early as 1958 (McCarthy). However very few attempts were made in creating such systems until the late 1970s. Expert systems providing a major impetus in this area.

There are two basic approaches to advice taking:

- Take high level, abstract advice and convert it into rules that can guide performance elements of the system. *Automate all aspects of advice taking*
- *Develop sophisticated tools* such as knowledge base editors and debugging. These are used to aid an expert to translate his expertise into detailed rules. Here the expert is an *integral* part of the learning system. Such tools are important in *expert systems* area of AI.

#### Automated Advice Taking

The following steps summarise this method:

#### Request

-- This can be simple question asking about general advice or more complicated by identifying shortcomings in the knowledge base and asking for a remedy.

#### Interpret

-- Translate the advice into an *internal representation*.

#### Operationalise

-- Translated advice may still not be usable so this stage seeks to provide a representation that can be used by the performance element.

#### Integrate

-- When knowledge is added to the knowledge base care must be taken so that bad side-effects are avoided.



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

E.g. Introduction of redundancy and contradictions.

Evaluate

-- The system must assess the new knowledge for errors, contradictions *etc.*

The steps can be iterated.

#### Knowledge Base Maintenance

Instead of automating the five steps above, many researchers have instead assembled tools that aid the development and maintenance of the knowledge base.

Many have concentrated on:

- Providing intelligent editors and flexible representation languages for integrating new knowledge.
- Providing debugging tools for evaluating, finding contradictions and redundancy in the existing knowledge base.

EMYCIN is an example of such a system.

#### Example Learning System - FOO

Learning the game of hearts

FOO (First Operational Operationaliser) tries to convert high level advice (principles, problems, methods) into effective executable (LISP) procedures.

Hearts:

- Game played as a series of *tricks*.
- One player - who has the *lead* - plays a card.
- Other players follow in turn and play a card.
  - The player must follow suit.
  - If he cannot he play any of his cards.
- The player who plays the highest value card *wins* the trick and the lead.
- The winning player takes the cards played in the trick.
- The *aim* is to avoid taking points. Each heart counts as one point the queen of spades is worth 13 points.
- The winner is the person that after all tricks have been played has the lowest points score.

Hearts is a game of partial information with no known algorithm for winning.

Although the possible situations are numerous general advice can be given such as:

- Avoid taking points.
- Do not lead a high card in suit in which an opponent is void.
- If an opponent has the queen of spades try to flush it.

In order to receive advice a human must convert into a FOO representation (LISP clause)

(avoid (take-points me) (trick))

FOO *operationalises* the advice by translating it into expressions it can use in the game. It can *UNFOLD* avoid and then trick to give:

(achieve (not (during

(scenario

(each p1 (players) (play-card p1))

(take-trick (trick-winner)))

(take-points me))))

However the advice is still not *operational* since it depends on the outcome of trick which is generally not known. Therefore FOO uses *case analysis* (on the during expression) to determine which steps could case one to take points. Step 1 is ruled out and step 2's take-points is UNFOLDED:

(achieve (not (exists c1 (cards-played)

(exists c2 (point-cards)

(during (take (trick-winner) c1)





तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

(take me c2))))))

FOO now has to decide: Under what conditions does (take me c2) occur during (take (trick-winner) c1).

A technique, called *partial matching*, hypothesises that points will be taken if me = trick-winner and c2 = c1. We can reduce our expression to:

(achieve (not (and (have-points(card-played))

(= (trick-winner) me ))))

This not quite enough a this means *Do not win trick that has points*. We do not know who the trick-winner is, also we have not said anything about how to play in a trick that has point led in the suit. After a few more steps to achieve this FOO comes up with:

(achieve (>= (and (in-suit-led(card-of me))

(possible (trick-has-points)))

(low(card-of me)))

FOO had an initial knowledge base that was made up of:

- basic domain concepts such as trick, hand, deck suits, avoid, win *etc.*
- Rules and behavioural constraints -- general rules of the game.
- Heuristics as to how to UNFOLD.

FOO has 2 basic shortcomings:

- It lacks a control structure that could apply operationalisation automatically.
- It is specific to hearts and similar tasks.

### Learning by Problem Solving

There are three basic methods in which a system can learn from its own experiences.

#### Learning by Parameter Adjustment

Many programs rely on an evaluation procedure to summarise the state of search *etc.* Game playing programs provide many examples of this.

However, many programs have a static evaluation function.

In learning a slight modification of the formulation of the evaluation of the problem is required.

Here the problem has an evaluation function that is represented as a polynomial of the form such as:

$$c_1t_1 + c_2t_2 + c_3t_3 + \dots$$

The  $t$  terms a values of features and the  $c$  terms are weights.

In designing programs it is often difficult to decide on the exact value to give each weight initially.

So the basic idea of idea of *parameter adjustment* is to:

- Start with some estimate of the correct weight settings.
- Modify the weight in the program on the basis of accumulated experiences.
- Features that appear to be good predictors will have their weights increased and bad ones will be decreased.

Samuel's Checkers programs employed 16 such features at any one time chosen from a pool of 38.

#### Learning by Macro Operators

The basic idea here is similar to Rote Learning:

*Avoid expensive recomputation*

*Macro-operators* can be used to group a whole series of actions into one.

For example: Making dinner can be described a lay the table, cook dinner, serves dinner. We could treat laying the table as on action even though it involves a sequence of actions.

The STRIPS problem-solving employed macro-operators in it's learning phase.

Consider a blocks world example in which ON(C,B) and ON(A, TABLE) are true.

STRIPS can achieve ON(A,B) in four steps:



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

UNSTACK(C,B), PUTDOWN(C), PICKUP(A), STACK(A,B)

STRIPS now builds a macro-operator MACROP with preconditions ON(C,B), ON(A,TABLE), postconditions ON(A,B), ON(C,TABLE) and the four steps as its body.

MACROP can now be used in future operation.

But it is not very general. The above can be easily generalised with variables used in place of the blocks.

However generalisation is not always that easy (See Rich and Knight).

### Learning by Chunking

*Chunking* involves similar ideas to Macro Operators and originates from psychological ideas on memory and problem solving.

The computational basis is in production systems (studied earlier).

SOAR is a system that use production rules to represent its knowledge. It also employs chunking to learn from experience.

Basic Outline of SOAR's Method

- SOAR solves problems it fires productions these are stored in *long term memory*.
- Some firings turn out to be more useful than others.
- When SOAR detects are useful sequence of firings, it creates *chunks*.
- A *chunk* is essentially a large production that does the work of an entire sequence of smaller ones.
- Chunks may be generalised before storing.

### Learning from example-induction:-

#### Inductive Learning

This involves the process of *learning by example* -- where a system tries to induce a general rule from a set of observed instances.

This involves classification -- assigning, to a particular input, the name of a class to which it belongs. Classification is important to many problem solving tasks.

A learning system has to be capable of evolving its own class descriptions:

- Initial class definitions may not be adequate.
- The world may not be well understood or rapidly changing.

The task of constructing class definitions is called *induction* or *concept learning*

A Blocks World Learning Example -- Winston (1975)

- The goal is to construct representation of the definitions of concepts in this domain.
- Concepts such a house - brick (rectangular block) with a wedge (triangular block) suitably placed on top of it, tent - 2 wedges touching side by side, or an arch - two non-touching bricks supporting a third wedge or brick, were learned.
- The idea of *near miss* objects -- similar to actual instances was introduced.
- Input was a line drawing of a blocks world structure.
- Input processed (see VISION Sections later) to produce a semantic net representation of the structural description of the object (Fig. below)

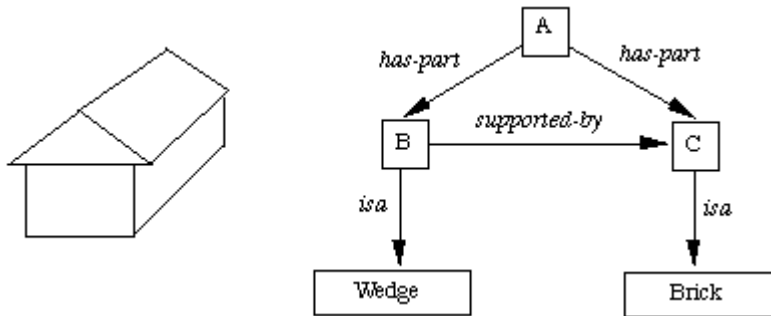


Fig. House object and semantic net

- Links in network include *left-of*, *right-of*, *does-not-marry*, *supported-by*, *has-part*, and *isa*.
- The *marry* relation is important -- two objects with a common touching edge are said to marry. Marrying is assumed unless *does-not-marry* stated.

There are three basic steps to the problem of concept formulation:

1. Select one known instance of the concept. Call this the *concept definition*.
2. Examine definitions of other known instance of the concept. *Generalise* the definition to include them.
3. Examine descriptions of *near misses*. *Restrict* the definition to *exclude* these.

Both steps 2 and 3 rely on comparison and both similarities and differences need to be identified.

Structural concept learning systems are not without their problems.

The biggest problem is that the *teacher* must guide the system through carefully chosen sequences of examples.

In Winston's program the order of the process is important since new links are added as and when new knowledge is gathered.

The concept of *version spaces* aims is insensitive to order of the example presented.

To do this instead of evolving a single concept description a set of possible descriptions are maintained. As new examples are presented the set evolves as a process of new instances and near misses.


We will assume that each *slot* in a version space description is made up of a set of predicates that do not negate other predicates in the set -- *positive literals*.

Indeed we can represent a description as a frame bases representation with several slots or indeed use a more general representation. For the sake of simplifying the discussion we will keep to simple representations.

If we keep to the above definition the Mitchell's *candidate elimination algorithm* is the best known algorithm.

Let us look at an example where we are presented with a number of playing cards and we need to learn if the card is *odd and black*.

We already know things like *red*, *black*, *spade*, *club*, *even card*, *odd card* etc.

So the  is *red* card, an *even* card and a *heart*.

This illustrates one of the keys to the version space method *specificity*:

- Conjunctive concepts in the domain can be partially ordered by specificity.
- In this Cards example the concept *black* is less specific than *odd black* or *spade*.
- *odd black* and *spade* are incomparable since neither is more (or less) specific.
- *Black* is more specific than *any card*, *any 8* or *any odd card*

The training set consist of a collection of cards and for each we are told whether or not it is in the *target set* -- *odd black*



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

The training set is dealt with *incrementally* and a list of most and least specific concepts consistent with training instances are maintained.

Let us see how can learn from a sample input set:

- Initially the most specific concept consistent with the data is the empty set. The least specific concept is the set of all cards.
- Let the  $A♠$  be the first card in the sample set. We are told that this is *odd black*.
- So the most specific concept is  $A♠$  alone the least is still all our cards.
- Next card  $8♠$ : we need to modify our most specific concept to indicate the generalisation of the set something like "odd and black cards". Least remains unchanged.
- Next card  $4♥$ : Now we can modify the least specific set to exclude the  $4♥$ . As more exclusion are added we will generalise this to all black cards and all odd cards.
- NOTE that negative instances cause least specific concepts to become more specific and positive instances similarly affect the most specific.
- If the two sets become the same set then the result is guaranteed and the target concept is met.

The Candidate Elimination Algorithm

Let us now formally describe the algorithm.

Let  $G$  be the set of most general concepts. Let  $S$  be the set of most specific concepts.

Assume: We have a common representation language and we a given a set of negative and positive training examples.

Aim: A concept description that is consistent with all the positive and *none* of the negative examples.

Algorithm:

- Initialise  $G$  to contain one element -- the *null* description, all features are variables.
- Initialise  $S$  to contain one element the first positive example.
- Repeat
  - Input the next training example
  - If a *positive example* -- first remove from  $G$  any descriptions that do not cover the example. Then update  $S$  to contain the most specific set of descriptions in the version space that cover the example and the current element set of  $S$ . *I.e. Generalise* the elements of  $S$  as little as possible so that they cover the new training example.
  - If a *negative example* -- first remove from  $S$  any descriptions that cover the example. Then update  $G$  to contain the most general set of descriptions in the version space that do not cover the example. *I.e. Specialise* the elements of  $S$  as little as possible so that negative examples are no longer covered in  $G$ 's elements.
- until  $S$  and  $G$  are both singleton sets.
- If  $S$  and  $G$  are identical output their value.
- $S$  and  $G$  are different then training sets were inconsistent.

Let us now look at the problem of learning the concept of a *flush* in poker where all five cards are of the same suit.

$(5♣, 7♣, 8♣, J♣, K♣)$

Let the first example be positive:

Then

we

set

$$G = \{(\varnothing_1, \varnothing_2, \varnothing_3, \varnothing_4, \varnothing_5)\}$$

$$S = \{(5♣, 7♣, 8♣, J♣, K♣)\}$$

$(5♣, 5♥, 8♦, J♥, A♠)$

No the second example is negative:

We must specialise  $G$  (only to current set):

$$G = \{ (x_1, x_2 = 7\clubsuit, x_3, x_4, x_5), \\ (x_1, x_2, x_3 = 8\clubsuit, x_4, x_5), \\ (x_1, x_2, x_3, x_4 = J\clubsuit, x_5), \\ (x_1, x_2, x_3, x_4, x_5 = K\clubsuit) \}$$

$S$  is unaffected.

$$(5\clubsuit, 6\clubsuit, 9\clubsuit, 10\clubsuit, Q\clubsuit)$$

Our third example is positive:

Firstly remove inconsistencies from  $G$  and then generalise  $S$ :

$$G = \{ (x_1, x_2 = \clubsuit, x_3, x_4, x_5), \\ (x_1, x_2, x_3 = \clubsuit, x_4, x_5), \\ (x_1, x_2, x_3, x_4 = \clubsuit, x_5), \\ (x_1, x_2, x_3, x_4, x_5 = \clubsuit) \}$$

$$S = \{ (x_1 = 5\clubsuit, x_2 = \clubsuit, x_3 = \clubsuit, x_4 = \clubsuit, x_5 = \clubsuit) \}$$

$$(A\heartsuit, 6\heartsuit, 9\heartsuit, 10\heartsuit, Q\heartsuit)$$

Our fourth example is also positive:

Once more remove inconsistencies from  $G$  and then generalise  $S$ :

$$G = \{ (x_1 = x_2 = x_3 = x_4 = x_5 = \text{same suit}, x_2, x_3, x_4, x_5), \\ S = \{ (x_1 = x_2 = x_3 = x_4 = x_5 = \text{same suit}, x_2, x_3, x_4, x_5) \}$$

- We can continue generalising and specialising
- We have taken a few big jumps in the flow of specialising/generalising in this example. Many more training steps usually required to reach this conclusion.
- It might be hard to spot trend of same suit *etc.*

### Decision Trees

Quinlan in his ID3 system (1986) introduced the idea of decision trees. ID3 is a program that can build trees automatically from given positive and negative instances.

Basically each leaf of a *decision tree* asserts a positive or negative concept. To classify a particular input we start at the top and follow assertions down until we reach an answer (Fig below)

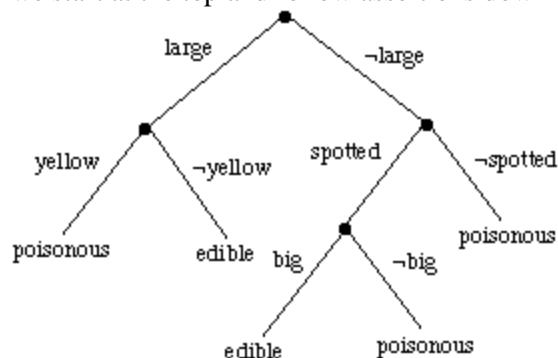


Fig. *Edible Mushroom* decision tree

Building decision trees



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

- ID3 uses an iterative method.
- Simple trees preferred as more accurate classification is afforded.
- A random choice of samples from training set chosen for initial assembly of tree -- the *window* subset.
- Other training examples used to test tree.
- If all examples classified correctly stop.
- Otherwise add a number of training examples to *window* and start again.

Adding new nodes

When assembling the tree we need to choose when to add a new node:

- Some attributes will yield more information than others.
- Adding a new node might be useless in the overall classification process.
- Sometimes attributes will separate training instances into subsets whose members share a common label. Here branching can be terminates and a leaf node assigned for the whole subset.

Decision tree advantages:

- Quicker than version spaces when concept space is large.
- Disjunction easier.

Disadvantages:

- Representation not natural to humans -- a decision tree may find it hard to explain its classification.

### **Explanation Based Learning (EBL):-**

Humans appear to learn quite a lot from one example.

Basic idea: Use results from one examples problem solving effort next time around.

An EBL accepts 4 kinds of input:

A training example

-- what the learning *sees* in the world.

A goal concept

-- a high level description of what the program is supposed to learn.

A operational criterion

-- a description of which concepts are usable.

A domain theory

-- a set of rules that describe relationships between objects and actions in a domain.

From this EBL computes a generalisation of the training example that is sufficient not only to describe the goal concept but also satisfies the operational criterion.

This has two steps:

Explanation

-- the domain theory is used to prune away all unimportant aspects of the training example with respect to the goal concept.

Generalisation

-- the explanation is generalised as far possible while still describing the goal concept.

### **EBL example**

Goal: To get to Brecon -- a picturesque welsh market town famous for its mountains (beacons) and its Jazz festival.

The training data is:

near(Cardiff, Brecon), airport(Cardiff)

The Domain Knowledge is:

near(x,y)  $\wedge$  holds(loc(x),s)  $\rightarrow$  holds(loc(y), result(drive(x,y),s)) airport(z)  $\rightarrow$  loc(z), result(fly(z),s)))



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

In this case operational criterion is: We must express concept definition in pure description language syntax. Our goal can expressed as follows:

holds(loc(Brecon),s) -- find some situation *s* for this holds.

We can prove this holds with *s* defined by:

result(drive(Cardiff,Brecon), result(fly(Cardiff), s'))

We can fly to Cardiff and then drive to Brecon.

If we analyse the proof (say with an ATMS). We can learn a few general rules from it.

Since Brecon appears in query and binding we could abstract it to give:

holds(loc(x),drive(Cardiff,x), result(fly(Cardiff), s'))

but this not quite right - we cannot get everywhere by flying to Cardiff.

Since Brecon appears in the database when we abstract things we must explicitly record the use of the fact:

near(Cardiff,x)  $\rightarrow$  holds(loc(x),drive(Cardiff,x), result(fly(Cardiff), s'))

This states if *x* is near Cardiff we can get to it by flying to Cardiff and then driving. We have *learnt* this general rule.

We could also abstract out Cardiff instead of Brecon to get:

near(Brecon,x)  $\wedge$  airport(x)  $\rightarrow$  holds(loc(Brecon), result(drive(x,Brecon), result(fly(x),s')))

This states we can get to Brecon by flying to another nearby airport and driving from there.

We could add airport(Swansea) and get an alternative means of travel plan.

Finally we could actually abstract out both Brecon and Cardiff to get a general plan:

near(x,y)  $\wedge$  airport(y)  $\rightarrow$  holds(loc(y), result(drive(x,y),result(fly(x),s')))

### Definition: Generalization

Generalization is the ability of a machine learning algorithm to perform accurately on new, unseen examples after training on a finite data set. The core objective of a learner is to generalize from its experience. The training examples from its experience come from some generally unknown probability distribution and the learner has to extract from them something more general, something about that distribution, that allows it to produce useful answers in new cases.

Machine learning, knowledge discovery in databases (KDD) and data mining

These three terms are commonly confused, as they often employ the same methods and overlap strongly.

They can be roughly separated as follows:

- Machine learning focuses on the prediction, based on *known* properties learned from the training data
- Data mining (which is the analysis step of Knowledge Discovery in Databases) focuses on the discovery of (previously) *unknown* properties on the data

However, these two areas overlap in many ways: data mining uses many machine learning methods, but often with a slightly different goal in mind. On the other hand, machine learning also employs data mining methods as "unsupervised learning" or as a preprocessing step to improve learner accuracy. Much of the confusion between these two research communities (which do often have separate conferences and separate journals, ECML PKDD being a major exception) comes from the basic assumptions they work with: in machine learning, the performance is usually evaluated with respect to the ability to *reproduce known* knowledge, while in KDD the key task is the discovery of previously *unknown* knowledge. Evaluated with respect to known knowledge, an uninformed (unsupervised) method will easily be outperformed by supervised methods, while in a typical KDD task, supervised methods cannot be used due to the unavailability of training data.



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

## Human interaction

Some machine learning systems attempt to eliminate the need for human intuition in data analysis, while others adopt a collaborative approach between human and machine. Human intuition cannot, however, be entirely eliminated, since the system's designer must specify how the data is to be represented and what mechanisms will be used to search for a characterization of the data.

## Algorithm types

Machine learning algorithms can be organized into a taxonomy based on the desired outcome of the algorithm.

- Supervised learning generates a function that maps inputs to desired outputs (also called labels, because they are often provided by human experts labeling the training examples). For example, in a classification problem, the learner approximates a function mapping a vector into classes by looking at input-output examples of the function.
- Unsupervised learning models a set of inputs, like clustering. See also data mining and knowledge discovery.
- Semi-supervised learning combines both labeled and unlabeled examples to generate an appropriate function or classifier.
- Reinforcement learning learns how to act given an observation of the world. Every action has some impact in the environment, and the environment provides feedback in the form of rewards that guides the learning algorithm.
- Transduction tries to predict new outputs based on training inputs, training outputs, and test inputs.
- Learning to learn learns its own inductive bias based on previous experience.

## Theory

The computational analysis of machine learning algorithms and their performance is a branch of theoretical computer science known as computational learning theory. Because training sets are finite and the future is uncertain, learning theory usually does not yield guarantees of the performance of algorithms. Instead, probabilistic bounds on the performance are quite common.

In addition to performance bounds, computational learning theorists study the time complexity and feasibility of learning. In computational learning theory, a computation is considered feasible if it can be done in polynomial time. There are two kinds of time complexity results. Positive results show that a certain class of functions can be learned in polynomial time. Negative results show that certain classes cannot be learned in polynomial time.

There are many similarities between machine learning theory and statistics, although they use different terms.

## Reinforcement learning

Reinforcement learning is concerned with how an *agent* ought to take *actions* in an *environment* so as to maximize some notion of long-term *reward*. Reinforcement learning algorithms attempt to find a *policy* that maps *states* of the world to the actions the agent ought to take in those states. Reinforcement learning differs from the supervised learning problem in that correct input/output pairs are never presented, nor sub-optimal actions explicitly corrected.

## Representation learning

Several learning algorithms, mostly unsupervised learning algorithms, aim at discovering better representations of the inputs provided during training. Classical examples include principal components analysis and cluster analysis. Representation learning algorithms often attempt to preserve the information in their input but transform it in a way that makes it useful, often as a pre-processing step before performing classification or predictions, allowing to reconstruct the inputs coming from the unknown data generating distribution, while not being necessarily faithful for configurations that are implausible under





तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

that distribution. Manifold learning algorithms attempt to do so under the constraint that the learned representation is low-dimensional. Sparse coding algorithms attempt to do so under the constraint that the learned representation is sparse (has many zeros). Deep learning algorithms discover multiple levels of representation, or a hierarchy of features, with higher-level, more abstract features defined in terms of (or generating) lower-level features. It has been argued that an intelligent machine is one that learns a representation that disentangles the underlying factors of variation that explain the observed data.

#### Sparse Dictionary Learning

In the learning area, sparse dictionary learning is one of the most popular methods, and has gained a huge success in lots of applications. In sparse dictionary learning, a datum is represented as a linear combination of basis functions, and the coefficients are assumed to be sparse. Let  $x$  be a  $d$ -dimensional datum,  $D$  be a  $d$  by  $n$  matrix, where each column of  $D$  represent a basis function.  $r$  is the coefficient to represent  $x$  using  $D$ . Mathematically, sparse dictionary learning means the following  $x \approx D \times r$  where  $r$  is sparse. Generally speaking,  $n$  is assumed to be larger than  $d$  to allow the freedom for a sparse representation.

Sparse dictionary learning has been applied in different context. In classification, the problem is to determine whether a new data belongs to which classes. Suppose we already build a dictionary for each class, then a new data is associate to the class such that it is best sparsely represented by the corresponding dictionary. People also applied sparse dictionary learning in image denoising. The key idea is that clean image path can be sparsely represented by a image dictionary, but the noise cannot. User can refer to if interested.

## UNIT – IV

### Expert System: Introduction:-

Expert systems are computer programs that mimic the special abilities of human experts. They use an accumulation of domain-specific knowledge: information collected through long experience with some particular problem-solving situation. Expert systems are constructed by taking this knowledge (typically from expert humans) and activating it with simple "if-then" rule systems. A beginner can then make decisions like an expert, simply by answering a series of simple questions.

An early, successful example of an expert system was the MYCIN system, used to help doctors diagnose medical conditions and find appropriate treatments. MYCIN asks about the patient's symptoms then guides the doctor through a series of diagnostic tests, ultimately recommending a medicine or treatment for the disorder.

What are expert systems? What are some successful uses of expert systems?

Expert systems running on computers have succeeded in a variety of domains. They have located oil deposits for oil companies. They guide stock purchases. They diagnosis car problems electronically. They are used in shopping malls to aid people matching make-up colors to hair or clothes.

Why did early dreams of making money on expert systems fail to come true?

When expert systems were a new concept, in the early 1980s, investment companies saw great commercial potential in them. Many new companies were formed, expecting to make a fortune by generating computer-based expertise. But the bubble burst. By the late 80s there was a "shake-out" and many of the companies formed earlier in the decade failed. Part of the problem was that expert systems performed erratically (see the discussion of "brittleness" below.) But another problem was that expert



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
**Institute of Management & Technology**  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

systems turned out to be relatively cheap to produce. That was bad news for people who wanted to make money selling them.

Here is one concrete example. A program for helping bank officers decide whether to grant mortgages in 1980 cost up to \$100,000 and required a mainframe computer. By 1988 it cost about \$100 and ran on any desktop computer. That was good for new mortgage lending companies, but it was not good for companies trying to make money-selling expert systems to banks (Moskowitz, 1988).

Why did some users become disillusioned with expert systems?

As expert systems became cheap and widely available, their shortcomings became more obvious, too. Many users tried them out briefly and became disillusioned with the amount of work required to make them work. After all, an expert system is supposed to mimic the knowledge base of a human expert, but a human expert spends thousands of hours accumulating knowledge. To imitate this expertise, somebody must spend a large amount of time entering rules into the computer, mostly of the form, "If you see X, do Y, but only if Z is true...."

The toughest part of setting up an expert system is to accumulate all the correct and most-needed knowledge and put it into the database. Humans must invest time and energy to do this, before the system will work. Moreover, these rules are different for each problem-solving domain. That is what "domain-specific knowledge" is all about: knowledge specific to one domain. So, in order to make an expert system work on a computer, one must first hire an expert to spell out the rules and exceptions to the rules, then spend a lot of time entering them into a database and debugging the program to eliminate errors and unpleasant surprises.

What does it mean to say that expert systems were "brittle"?

Expert humans with common sense must oversee even a commercially available expert system, fine-tuned for a specific domain. For example, no doctor in his right mind would blindly follow all the advice of the MYCIN system without thinking independently about whether the diagnosis "made sense" and was safe and appropriate for the patient. To do otherwise would be to risk malpractice suits. Similarly, a stockbroker would be foolish to follow every recommendation of a stock-trading program without exercising common sense. Expert systems are notorious for occasionally producing silly or absurd recommendations. Somebody has to be checking the performance of the system, to catch such problems.

Researchers have a word for the tendency of programs to function only in a limited, predictable context. Such behavior is called brittle. To be "brittle" means to work only as long as problems are set up in ways that are anticipated. A brittle system may "break" (produce incorrect results) as soon as odd or unanticipated situations are encountered. Expert systems are usually brittle until they have been field-tested in many different situations. That is the only way to detect unanticipated problems. So an expert system— even after having many thousands of rules entered into it— may require human experts to act as baby-sitters for many years while it is field tested, which defeats the whole idea of turning important decisions over to computers.

Expert system: a computer system or program that uses artificial intelligence techniques to solve problems that ordinarily require a knowledgeable human. The method used to construct such systems, knowledge engineering, extracts a set of rules and data from an expert or experts through extensive questioning. This material is then organized in a format suitable for representation in a computer and a set of tools for inquiry, manipulation, and response is applied. While such systems do not often replace the human experts, they can serve as useful adjuncts or assistants. Among some of the successful expert systems developed are INTERNIST, a medical diagnosis tool that contains nearly 100,000 relationships between symptoms and diseases, and PROSPECTOR, an aid to geologists in interpreting mineral data.



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

A computer program that contains a knowledge base and a set of algorithms or rules that infer new facts from knowledge and from incoming data. An expert system is an artificial intelligence application that uses a knowledge base of human expertise to aid in solving problems. The degree of problem solving is based on the quality of the data and rules obtained from the human expert. Expert systems are designed to perform at a human expert level. In practice, they will perform both well below and well above that of an individual expert. The expert system derives its answers by running the knowledge base through an inference engine, a software program that interacts with the user and processes the results from the rules and data in the knowledge base.

Expert systems are used in applications such as medical diagnosis, equipment repair, investment analysis, financial, estate and insurance planning, route scheduling for delivery vehicles, contract bidding, counseling for self-service customers, production control and training.

Example of Expert system

MYCIN: medical diagnosis using production rules

MYCIN was the first well known medical expert system developed by Shortliffe at Stanford University to help doctors, not expert in antimicrobial drugs, prescribe such drugs for blood infections. MYCIN has three sub-systems:

1. Consultation system
2. Explanation System
3. Rule Acquisition system

Problems that occur with drug prescription

- Non-expert doctors don't go through this process - prescribe habitual choice
- Over prescription (e.g. For viral illnesses) estimated at time at 10-20 times necessary
- Wastes money
- Can make for resistant strains of organisms

1. Mycin's Consultation System

- Works out possible organisms and suggests treatments

Static and dynamic data structures

- Static data structures
- These store medical knowledge not suitable for storage as inferential rules: includes lists of organisms, knowledge tables with features of bacteria types, list of parameters
- Parameters = features of patients, bacterial cultures, drugs
- Parameters can be Y/N (e.g. FEBRILE), single value (e.g. IDENTITY - if it's salmonella it can't be another organism as well) or multi-value (e.g. INFECT - patient can have more than one infection)
- Parameter properties
- EXPECT range of possible values
- PROMPT English sentence to elicit response  
LABDATA can be known for certain from Lab data  
LOOKAHEAD lists rules mentioning the parameter in their premise (e.g. a rule might need to know whether or not a patient is febrile)  
UPDATED-BY lists rules mentioning the parameter in their action (i.e. they may draw a conclusion about the value of the parameter, such as the IDENTITY parameter)
- Dynamic data structures store information about the evolving case - the patient details, possible diagnoses, rules consulted:
- Example piece of dynamic data:
- To evaluate the premise of the rule mentioned above:



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

- (\$AND (SAME ORGANISM-1 GRAM GRAMNEG)
- (SAME ORGANISM-1 MORPH ROD)
- (SAME ORGANISM-1 AIR AEROBIC))
- from following data
- ORGANISM-1
- GRAM = (GRAMNEG 1.0)
- MORPH = (ROD 0.8) (COCCUS 0.2)
- AIR = (AEROBIC 0.6) (FACUL 0.4)
- Total value of whole expression is lowest certainty value = 0.6
- Conclusion is therefore:
- (CONCLUDE ORGANISM-1 is enterobacteriaceae 0.48 (= 0.6 (CF of premise) x 0.8 (CF of rule))
- (In disjunctive premise, total value of whole expression is highest certainty value)

Control structure

MYCIN first attempts to create a "patient context" containing information about the case, then tries to compile a list of therapies for the context.

It uses a backward chaining mechanism, reasoning back from the goals it wants to prove to the data it has, rather than vice versa. The overall goal is "compile a list of therapies".

Questions can be prompted by the invocation of rules, to find out necessary data, to avoid unnecessary questions.

## 2. The Explanation System

MYCIN can answer questions about HOW a conclusion was reached and WHY a question was asked, either after a consultation or while it is going on.

It does this by manipulating its record of the rules it invoked, the goal it was trying to achieve, the information it was trying to discover.

Can also answer general questions (e.g. what would you prescribe for organism X?) by consulting its static data structures.

## 3. The Rule Acquisition System

Experts can enter new rules or edit existing rules. The system automatically adds the new rule to the LOOKAHEAD list for all parameters mentioned in its premise, and to the UPDATED-BY list of all parameters mentioned in its action.

### **Representing using domain specific knowledge:-**

Domain knowledge is valid knowledge used to refer to an area of human endeavour, an autonomous computer activity, or other specialized discipline.

Specialists and experts use and develop their own domain knowledge. If the concept *domain knowledge* or domain expert is used, we emphasize a specific domain which is an object of the discourse/interest/problem.

Knowledge capture

In software engineering *domain knowledge* is knowledge about the environment in which the target system operates, for example, software agents. Domain knowledges are important, because it usually must be learned from software users in the domain (as domain specialists/experts), rather than from software developers. Experts' domain knowledge (frequently informal and ill-structured) is transformed in computer programs and active data, for example in a set of rules in knowledge bases, by knowledge engineers.



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
**Institute of Management & Technology**  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

Communicating between end-users and software developers is often difficult. They must find a common language to communicate in. Developing enough shared vocabulary to communicate can often take a while.

The same knowledge can be included in different domain knowledge. Knowledge which may be efficient in every domain is called *domain-independent* knowledge, for example logics and mathematics. Operations on domain knowledge are performed by meta-knowledge. Domain Knowledge is the knowledge of a particular stream.

### **Expert system shells:-**

A shell is a piece of software that provides an interface for users of an operating system which provides access to the services of a kernel. However, the term is also applied very loosely to applications and may include any software that is "built around" a particular component, such as web browsers and email clients that are "shells" for HTML rendering engines. The name *shell* originates from shells being an outer layer of interface between the user and the internals of the operating system (the kernel).

Operating system shells generally fall into one of two categories: command-line and graphical. Command-line shells provide a command-line interface (CLI) to the operating system, while graphical shells provide a graphical user interface (GUI). In either category the primary purpose of the shell is to invoke or "launch" another program; however, shells frequently have additional capabilities such as viewing the contents of directories.

The relative merits of CLI- and GUI-based shells are often debated. CLI proponents claim that certain operations can be performed much faster under CLI shells than under GUI shells (such as moving files, for example). However, GUI proponents advocate the comparative usability and simplicity of GUI shells. The best choice is often determined by the way in which a computer will be used. On a server mainly used for data transfers and processing with expert administration, a CLI is likely to be the best choice. However, a GUI would be more appropriate for a computer to be used for image or video editing and the development of the above data.

In expert systems, a shell is a piece of software that is an "empty" expert system without the knowledge base for any particular application

The E.S shell simplifies the process of creating a knowledge base. It is the shell that actually processes the information entered by a user relates it to the concepts contained in the knowledge base and provides an assessment or solution for a particular problem. Thus E.S shell provides a layer between the user interface and the computer O.S to manage the input and output of the data. It also manipulates the information provided by the user in conjunction with the knowledge base to arrive at a particular conclusion.

### **LISP , AI Programming Language:-**

Lisp was invented by John McCarthy in 1958 while he was at the Massachusetts Institute of Technology (MIT). McCarthy published its design in a paper in *Communications of the ACM* in 1960, entitled "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I"<sup>[21]</sup> ("Part II" was never published). He showed that with a few simple operators and a notation for functions, one can build a Turing-complete language for algorithms.

Information Processing Language was the first AI language, from 1955 or 1956, and already included many of the concepts, such as list-processing and recursion, which came to be used in Lisp.



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

McCarthy's original notation used bracketed "M-expressions" that would be translated into S-expressions. As an example, the M-expression `car[cons[A,B]]` is equivalent to the S-expression `(car (cons A B))`. Once Lisp was implemented, programmers rapidly chose to use S-expressions, and M-expressions were abandoned. M-expressions surfaced again with short-lived attempts of MLISP by Horace Enea and CGOL by Vaughan Pratt.

Lisp was first implemented by Steve Russell on an IBM 704 computer. Russell had read McCarthy's paper, and realized (to McCarthy's surprise) that the Lisp *eval* function could be implemented in machine code. The result was a working Lisp interpreter which could be used to run Lisp programs, or more properly, 'evaluate Lisp expressions.'

Two assembly language macros for the IBM 704 became the primitive operations for decomposing lists: `car` (Contents of the Address part of Register number) and `CDR` (Contents of the Decrement part of Register number). From the context, it is clear that the term "Register" is used here to mean "Memory Register", nowadays called "Memory Location". Lisp dialects still use `car` and `cdr` for the operations that return the first item in a list and the rest of the list respectively.

The first complete Lisp compiler, written in Lisp, was implemented in 1962 by Tim Hart and Mike Levin at MIT. This compiler introduced the Lisp model of incremental compilation, in which compiled and interpreted functions can intermix freely. The language used in Hart and Levin's memo is much closer to modern Lisp style than McCarthy's earlier code.

Lisp was a difficult system to implement with the compiler techniques and stock hardware of the 1970s. Garbage collection routines, developed by then-MIT graduate student Daniel Edwards, made it practical to run Lisp on general-purpose computing systems, but efficiency was still a problem. This led to the creation of Lisp machines: dedicated hardware for running Lisp environments and programs. Advances in both computer hardware and compiler technology soon made Lisp machines obsolete.

During the 1980s and 1990s, a great effort was made to unify the work on new Lisp dialects (mostly successors to Mac lisp like Zeta Lisp and NIL (New Implementation of Lisp)) into a single language. The new language, Common Lisp, was somewhat compatible with the dialects it replaced (the book Common Lisp the Language notes the compatibility of various constructs). In 1994, ANSI published the Common Lisp standard, "ANSI X3.226-1994 Information Technology Programming Language Common Lisp."

Connection to artificial intelligence

Since its inception, Lisp was closely connected with the artificial intelligence research community, especially on PDP-10 systems. Lisp was used as the implementation of the programming language Micro Planner which was used in the famous AI system SHRDLU. In the 1970s, as AI research spawned commercial offshoots, the performance of existing Lisp systems became a growing issue.

Genealogy and variants Over its fifty-year history, Lisp has spawned many variations on the core theme of an S-expression language. Moreover, each given dialect may have several implementations; for instance; there are more than a dozen implementations of Common Lisp.

Differences between dialects may be quite visible; for instance, Common Lisp and Scheme use different keywords to define functions. Within a dialect that is standardized, however, conforming implementations support the same core language, but with different extensions and libraries.

Historically significant dialects

- LISP 1 ó First implementation.
- LISP 1.5 ó First widely distributed version, developed by McCarthy and others at MIT. So named because it contained several improvements on the original "LISP 1" interpreter, but was not a major restructuring as the planned LISP 2 would be.



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

- Stanford LISP 1.6 ó This was a successor to LISP 1.5 developed at the Stanford AI Lab, and widely distributed to PDP-10 systems running the TOPS-10 operating system. It was rendered obsolete by Maclisp and InterLisp.
- MACLISP ó developed for MIT's Project MAC (no relation to Apple's Macintosh, nor to McCarthy), direct descendant of LISP 1.5. It ran on the PDP-10 and Multics systems. (MACLISP would later come to be called Maclisp, and is often referred to as MacLisp.)
- InterLisp ó developed at BBN Technologies for PDP-10 systems running the Tenex operating system, later adopted as a "West coast" Lisp for the Xerox Lisp machines as InterLisp-D. A small version called "InterLISP 65" was published for Atari's 6502-based computer line. For quite some time Maclisp and InterLisp were strong competitors.
- Franz Lisp ó originally a Berkeley project; later developed by Franz Inc. The name is a humorous deformation of the name "Franz Liszt", and does not refer to Allegro Common Lisp, the dialect of Common Lisp sold by Franz Inc., in more recent years.
- XLISP, which Auto LISP was based on.
- Standard Lisp and Portable Standard Lisp were widely used and ported, especially with the Computer Algebra System REDUCE.
- ZetaLisp, also known as Lisp Machine Lisp ó used on the Lisp machines, direct descendant of Maclisp. ZetaLisp had big influence on Common Lisp.
- LeLisp is a French Lisp dialect. One of the first Interface Builders was written in LeLisp.
- Common Lisp (1984), as described by *Common Lisp the Language* ó a consolidation of several divergent attempts (ZetaLisp, Spice Lisp, NIL, and S-1 Lisp) to create successor dialects<sup>[13]</sup> to Maclisp, with substantive influences from the Scheme dialect as well. This version of Common Lisp was available for wide-ranging platforms and was accepted by many as a de facto standard until the publication of ANSI Common Lisp (ANSI X3.226-1994).
- Dylan was in its first version a mix of Scheme with the Common Lisp Object System.
- EuLisp ó attempt to develop a new efficient and cleaned-up Lisp.
- ISLISP ó attempt to develop a new efficient and cleaned-up Lisp. Standardized as ISO/IEC 13816:1997 and later revised as ISO/IEC 13816:2007: *Information technology – Programming languages, their environments and system software interfaces – Programming language ISLISP*.
- IEEE Scheme ó IEEE standard, 117861990 (R1995)
- ANSI Common Lisp ó an American National Standards Institute (ANSI) standard for Common Lisp, created by subcommittee X3J13, chartered to begin with *Common Lisp: The Language* as a base document and to work through a public consensus process to find solutions to shared issues of portability of programs and compatibility of Common Lisp implementations. Although formally an ANSI standard, the implementation, sale, use, and influence of ANSI Common Lisp has been and continues to be seen worldwide.
- ACL2 or "A Computational Logic for Applicative Common Lisp", an applicative (side-effect free) variant of Common LISP. ACL2 is both a programming language in which you can model computer systems and a tool to help proving properties of those models.

Since 2000

After having declined somewhat in the 1990s, Lisp has recently experienced a resurgence of interest. Most new activity is focused around open source implementations of Common Lisp, and includes the development of new portable libraries and applications. A new print edition of *Practical Common Lisp* by Peter Seibel, a tutorial for new Lisp programmers, was published in 2005. It is available free online.

Many new Lisp programmers were inspired by writers such as Paul Graham and Eric S. Raymond to pursue a language others considered antiquated. New Lisp programmers often describe the language as an



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

eye-opening experience and claim to be substantially more productive than in other languages. This increase in awareness may be contrasted to the "AI winter" and Lisp's brief gain in the mid-1990s.

Dan Weinreb lists in his survey of Common Lisp implementations eleven actively maintained Common Lisp implementations. Sciener Common Lisp is a new commercial implementation forked from CMUCL with a first release in 2002.

The open source community has created new supporting infrastructure: CLiki is a wiki that collects Common Lisp related information, the Common Lisp directory lists resources, #lisp is a popular IRC channel (with support by a Lisp-written Bot), lispaste supports the sharing and commenting of code snippets, Planet Lisp collects the contents of various Lisp-related blogs, on LispForum users discuss Lisp topics, Lispjobs is a service for announcing job offers and there is a weekly news service, *Weekly Lisp News*. *Common-lisp.net* is a hosting site for open source Common Lisp projects.

50 years of Lisp (1958-2008) has been celebrated at LISP50@OOPSLA. There are regular local user meetings in Boston, Vancouver, and Hamburg. Other events include the European Common Lisp Meeting, the European Lisp Symposium and an International Lisp Conference.

The Scheme community actively maintains over twenty implementations. Several significant new implementations (Chicken, Gambit, Gauche, Ikarus, Larceny, Ypsilon) have been developed in the last few years. The Revised<sup>5</sup> Report on the Algorithmic Language Scheme standard of Scheme was widely accepted in the Scheme community. The Scheme Requests for Implementation process has created a lot of quasi standard libraries and extensions for Scheme. User communities of individual Scheme implementations continue to grow. A new language standardization process was started in 2003 and led to the R<sup>6</sup>RS Scheme standard in 2007. Academic use of Scheme for teaching computer science seems to have declined somewhat. Some universities are no longer using Scheme in their computer science introductory courses.

There are several new dialects of Lisp: Arc, Nu, and Clojure.

Major dialects

The two major dialects of Lisp used for general-purpose programming today are Common Lisp and Scheme. These languages represent significantly different design choices.

Common Lisp is a successor to MacLisp. The primary influences were Lisp Machine Lisp, MacLisp, NIL, S-1 Lisp, Spice Lisp, and Scheme. It has many of the features of Lisp Machine Lisp (a large Lisp dialect used to program Lisp Machines), but was designed to be efficiently implementable on any personal computer or workstation. Common Lisp has a large language standard including many built-in data types, functions, macros and other language elements, as well as an object system (Common Lisp Object System or shorter CLOS). Common Lisp also borrowed certain features from Scheme such as lexical scoping and lexical closures.

Scheme (designed earlier) is a more minimalist design, with a much smaller set of standard features but with certain implementation features (such as tail-call optimization and full continuations) not necessarily found in Common Lisp.

Scheme is a statically scoped and properly tail-recursive dialect of the Lisp programming language invented by Guy Lewis Steele Jr. and Gerald Jay Sussman. It was designed to have an exceptionally clear and simple semantics and few different ways to form expressions. A wide variety of programming paradigms, including imperative, functional, and message passing styles, find convenient expression in Scheme. Scheme continues to evolve with a series of standards (Revised<sup>n</sup> Report on the Algorithmic Language Scheme) and a series of Scheme Requests for Implementation.

Clojure is a dynamic programming dialect of Lisp that targets the Java Virtual Machine (and the CLR ). It is designed to be a general-purpose language, combining the approachability and interactive development of a scripting language with an efficient and robust infrastructure for multithreaded programming. Clojure is a compiled language, as it compiles directly to JVM bytecode, yet remains completely dynamic. Every





तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

feature supported by Clojure is supported at runtime. Clojure provides easy access to the Java frameworks, with optional type hints and type inference, to ensure that calls to Java can avoid reflection. In addition, Lisp dialects are used as scripting languages in a number of applications, with the most well-known being Emacs Lisp in the Emacs editor, Visual Lisp in AutoCAD, Nyquist in Audacity. The small size of a minimal but useful Scheme interpreter makes it particularly popular for embedded scripting. Examples include SIOD and TinyScheme, both of which have been successfully embedded in the GIMP image processor under the generic name "Script-fu". LIBREP, a Lisp interpreter by John Harper originally based on the Emacs Lisp language, has been embedded in the Sawfish window manager.<sup>[25]</sup> The Guile interpreter is used in GnuCash. Within GCC, the MELT plugin provides a Lisp-y dialect, translated into C, to extend the compiler by coding additional passes (in MELT).

#### Language innovations

Lisp was the first homoiconic programming language: the primary representation of program code is the same type of list structure that is also used for the main data structures. As a result, Lisp functions can be manipulated, altered or even created within a Lisp program without extensive parsing or manipulation of binary machine code. This is generally considered one of the primary advantages of the language with regard to its expressive power, and makes the language amenable to metacircular evaluation.

The ubiquitous *if-then-else* structure, now taken for granted as an essential element of any programming language, was invented by McCarthy for use in Lisp, where it saw its first appearance in a more general form (the *cond* structure). It was inherited by ALGOL, which popularized it.

Lisp deeply influenced Alan Kay, the leader of the research on Smalltalk, and then in turn Lisp was influenced by Smalltalk, by adopting object-oriented programming features (classes, instances, etc.) in the late 1970s.

Largely because of its resource requirements with respect to early computing hardware (including early microprocessors), Lisp did not become as popular outside of the AI community as FORTRAN and the ALGOL-descended C language. Newer languages such as Java and Python have incorporated some limited versions of some of the features of Lisp, but are necessarily unable to bring the coherence and synergy of the full concepts found in Lisp. Because of its suitability to ill-defined, complex, and dynamic applications, Lisp is currently enjoying some resurgence of popular interest.

#### Syntax and semantics

##### Symbolic expressions

Lisp is an expression-oriented language. Unlike most other languages, no distinction is made between "expressions" and "statements"; all code and data are written as expressions. When an expression is *evaluated*, it produces a value (in Common Lisp, possibly multiple values), which then can be embedded into other expressions. Each value can be any data type.

McCarthy's 1958 paper introduced two types of syntax: S-expressions (Symbolic expressions, also called "sexps"), which mirror the internal representation of code and data; and M-expressions (Meta Expressions), which express functions of S-expressions. M-expressions never found favor, and almost all Lisps today use S-expressions to manipulate both code and data.

The use of parentheses is Lisp's most immediately obvious difference from other programming language families. As a result, students have long given Lisp nicknames such as *Lost In Stupid Parentheses*, or *Lots of Irritating Superfluous Parentheses*. However, the S-expression syntax is also responsible for much of Lisp's power: the syntax is extremely regular, which facilitates manipulation by computer. However, the syntax of Lisp is not limited to traditional parentheses notation. It can be extended to include alternative notations. XMLisp, for instance, is a Common Lisp extension that employs the metaobject-protocol to integrate S-expressions with the Extensible Markup Language (XML).

The reliance on expressions gives the language great flexibility. Because Lisp functions are themselves written as lists, they can be processed exactly like data. This allows easy writing of programs which



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

manipulate other programs (metaprogramming). Many Lisp dialects exploit this feature using macro systems, which enables extension of the language almost without limit.

Lists

### AI Programming Language:-

A Lisp list is written with its elements separated by whitespace, and surrounded by parentheses. For example, (1 2 foo) is a list whose elements are three *atoms*: the values 1, 2, and foo. These values are implicitly typed: they are respectively two integers and a Lisp-specific data type called a "symbol", and do not have to be declared as such.

The empty list () is also represented as the special atom nil. This is the only entity in Lisp which is both an atom and a list.

Expressions are written as lists, using prefix notation. The first element in the list is the name of a *form*, i.e., a function, operator, macro, or "special operator" (see below.) The remainder of the list are the arguments. For example, the function list returns its arguments as a list, so the expression

```
(list '1 '2 'foo)
```

evaluates to the list (1 2 foo). The "quote" before the arguments in the preceding example is a "special operator" which prevents the quoted arguments from being evaluated (not strictly necessary for the numbers, since 1 evaluates to 1, etc.). Any unquoted expressions are recursively evaluated before the enclosing expression is evaluated. For example,

```
(list 1 2 (list 3 4))
```

evaluates to the list (1 2 (3 4)). Note that the third argument is a list; lists can be nested.

Operators

Arithmetic operators are treated similarly. The expression

```
(+ 1 2 3 4)
```

evaluates to 10. The equivalent under infix notation would be "1 + 2 + 3 + 4". Arithmetic operators in Lisp are variadic (or *n-ary*), able to take any number of arguments.

"Special operators" (sometimes called "special forms") provide Lisp's control structure. For example, the special operator if takes three arguments. If the first argument is non-nil, it evaluates to the second argument; otherwise, it evaluates to the third argument. Thus, the expression

```
(if nil
```

```
  (list 1 2 "foo")
```

```
  (list 3 4 "bar"))
```

evaluates to (3 4 "bar"). Of course, this would be more useful if a non-trivial expression had been substituted in place of nil.

Lambda expressions

Another special operator, lambda, is used to bind variables to values which are then evaluated within an expression. This operator is also used to create functions: the arguments to lambda are a list of arguments, and the expression or expressions to which the function evaluates (the returned value is the value of the last expression that is evaluated). The expression

```
(lambda (arg) (+ arg 1))
```

evaluates to a function that, when applied, takes one argument, binds it to arg and returns the number one greater than that argument. Lambda expressions are treated no differently from named functions; they are invoked the same way. Therefore, the expression

```
((lambda (arg) (+ arg 1)) 5)
```

evaluates to 6.

Atoms

In the original LISP there were two fundamental data types: atoms and lists. A list was a finite ordered sequence of elements, where each element is in itself either an atom or a list, and an atom was a number

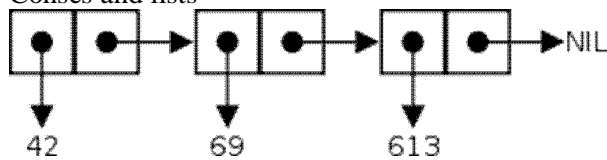


or a symbol. A symbol was essentially a unique named item, written as an Alphanumeric string in source code, and used either as a variable name or as a data item in symbolic processing. For example, the list (FOO (BAR 1) 2) contains three elements: the symbol FOO, the list (BAR 1), and the number 2.

The essential difference between atoms and lists was that atoms were immutable and unique. Two atoms that appeared in different places in source code but were written in exactly the same way represented the same object, whereas each list was a separate object that could be altered independently of other lists and could be distinguished from other lists by comparison operators.

As more data types were introduced in later Lisp dialects, and programming styles evolved, the concept of an atom lost importance. Many dialects still retained the predicate *atom* for legacy compatibility, defining it true for any object which is not a cons.

Conses and lists



Box-and-pointer diagram for the list (42 69 613)

A Lisp list is a singly linked list. Each cell of this list is called a *cons* (in Scheme, a *pair*), and is composed of two pointers, called the *car* and *cdr*. These are equivalent to the data and next fields discussed in the article *linked list*, respectively.

Of the many data structures that can be built out of cons cells, one of the most basic is called a *proper list*. A proper list is either the special nil (empty list) symbol, or a cons in which the car points to a datum (which may be another cons structure, such as a list), and the cdr points to another proper list.

If a given cons is taken to be the head of a linked list, then its car points to the first element of the list, and its cdr points to the rest of the list. For this reason, the car and cdr functions are also called first and rest when referring to conses which are part of a linked list (rather than, say, a tree).

Thus, a Lisp list is not an atomic object, as an instance of a container class in C++ or Java would be. A list is nothing more than an aggregate of linked conses. A variable which refers to a given list is simply a pointer to the first cons in the list. Traversal of a list can be done by "cdring down" the list; that is, taking successive cdrs to visit each cons of the list; or by using any of a number of higher-order functions to map a function over a list.

Because conses and lists are so universal in Lisp systems, it is a common misconception that they are Lisp's only data structures. In fact, all but the most simplistic Lisps have other data structures ó such as vectors (arrays), hash tables, structures, and so forth.

S-expressions represent lists

Parenthesized S-expressions represent linked list structures. There are several ways to represent the same list as an S-expression. A cons can be written in *dotted-pair notation* as (a . b), where a is the car and b the cdr. A longer proper list might be written (a . (b . (c . (d . nil)))) in dotted-pair notation. This is conventionally abbreviated as (a b c d) in *list notation*. An improper list<sup>[27]</sup> may be written in a combination of the two ó as (a b c . d) for the list of three conses whose last cdr is d (i.e., the list (a . (b . (c . d))) in fully specified form).

List-processing procedures

Lisp provides many built-in procedures for accessing and controlling lists. Lists can be created directly with the list procedure, which takes any number of arguments, and returns the list of these arguments.

(list 1 2 'a 3)



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

```
;Output: (1 2 a 3)
```

```
(list 1 '(2 3) 4)
```

```
;Output: (1 (2 3) 4)
```

Because of the way that lists are constructed from cons pairs, the `[[cons]]` procedure can be used to add an element to the front of a list. Note that the cons procedure is asymmetric in how it handles list arguments, because of how lists are constructed.

```
(cons 1 '(2 3))
```

```
;Output: (1 2 3)
```

```
(cons '(1 2) '(3 4))
```

```
;Output: ((1 2) 3 4)
```

The `[[append]]` procedure appends two (or more) lists to one another. Because Lisp lists are linked lists, appending two lists has asymptotic time complexity  $O(n)$

```
(append '(1 2) '(3 4))
```

```
;Output: (1 2 3 4)
```

```
(append '(1 2 3) '()) '(a) '(5 6))
```

```
;Output: (1 2 3 a 5 6)
```

Shared structure

Lisp lists, being simple linked lists, can share structure with one another. That is to say, two lists can have the same *tail*, or final sequence of conses. For instance, after the execution of the following Common Lisp code:

```
(setf foo (list 'a 'b 'c))
```

```
(setf bar (cons 'x (cdr foo)))
```

the lists `foo` and `bar` are `(a b c)` and `(x b c)` respectively. However, the tail `(b c)` is the same structure in both lists. It is not a copy; the cons cells pointing to `b` and `c` are in the same memory locations for both lists.

Sharing structure rather than copying can give a dramatic performance improvement. However, this technique can interact in undesired ways with functions that alter lists passed to them as arguments. Altering one list, such as by replacing the `c` with a `goose`, will affect the other:

```
(setf (third foo) 'goose)
```

This changes `foo` to `(a b goose)`, but thereby also changes `bar` to `(x b goose)` ó a possibly unexpected result. This can be a source of bugs, and functions which alter their arguments are documented as *destructive* for this very reason.

Aficionados of functional programming avoid destructive functions. In the Scheme dialect, which favors the functional style, the names of destructive functions are marked with a cautionary exclamation point, or "bang" ó such as `set-car!` (read *set car bang*), which replaces the `car` of a cons. In the Common Lisp dialect, destructive functions are commonplace; the equivalent of `set-car!` is named `rplaca` for "replace car." This function is rarely seen however as Common Lisp includes a special facility, `setf`, to make it easier to define and use destructive functions. A frequent style in Common Lisp is to write code functionally (without destructive calls) when prototyping, then to add destructive calls as an optimization where it is safe to do so.

Self-evaluating forms and quoting

Lisp evaluates expressions which are entered by the user. Symbols and lists evaluate to some other (usually, simpler) expression ó for instance, a symbol evaluates to the value of the variable it names; `(+ 2 3)` evaluates to 5. However, most other forms evaluate to themselves: if you enter 5 into Lisp, it returns 5.

Any expression can also be marked to prevent it from being evaluated (as is necessary for symbols and lists). This is the role of the quote special operator, or its abbreviation `'` (a single quotation mark). For instance, usually if you enter the symbol `foo` you will get back the value of the corresponding variable (or



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

an error, if there is no such variable). If you wish to refer to the literal symbol, you enter (quote foo) or, usually, 'foo.

Both Common Lisp and Scheme also support the *backquote* operator (known as *quasiquote* in Scheme), entered with the ``` character. This is almost the same as the plain quote, except it allows expressions to be evaluated and their values interpolated into a quoted list with the comma and comma-at operators. If the variable `snue` has the value `(bar baz)` then ``(foo ,snue)` evaluates to `(foo (bar baz))`, while ``(foo ,@snue)` evaluates to `(foo bar baz)`. The backquote is most frequently used in defining macro expansions.

Self-evaluating forms and quoted forms are Lisp's equivalent of literals. It may be possible to modify the values of (mutable) literals in program code. For instance, if a function returns a quoted form, and the code that calls the function modifies the form, this may alter the behavior of the function on subsequent iterations.

```
(defun should-be-constant ()  
  '(one two three))
```

```
(let ((stuff (should-be-constant)))  
  (setf (third stuff) 'bizarre)) ; bad!
```

`(should-be-constant)` ; returns `(one two bizarre)`

Modifying a quoted form like this is generally considered bad style, and is defined by ANSI Common Lisp as erroneous (resulting in "undefined" behavior in compiled files, because the file-compiler can coalesce similar constants, put them in write-protected memory, etc.).

Lisp's formalization of quotation has been noted by Douglas Hofstadter (in *Gödel, Escher, Bach*) and others as an example of the philosophical idea of self-reference.

#### Scope and closure

The modern Lisp family splits over the use of dynamic or static (aka lexical) scope. Clojure, Common Lisp and Scheme make use of static scoping by default, while Newlisp, Picolisp and the embedded languages in Emacs and AutoCAD use dynamic scoping.

#### List structure of program code

A fundamental distinction between Lisp and other languages is that in Lisp, the textual representation of a program is simply a human-readable description of the same internal data structures (linked lists, symbols, number, characters, etc.) as would be used by the underlying Lisp system.

Lisp macros operate on these structures. Because Lisp code has the same structure as lists, macros can be built with any of the list-processing functions in the language. In short, anything that Lisp can do to a data structure, Lisp macros can do to code. In contrast, in most other languages, the parser's output is purely internal to the language implementation and cannot be manipulated by the programmer. Macros in C, for instance, operate on the level of the *preprocessor*, before the parser is invoked, and cannot re-structure the program code in the way Lisp macros can.

In simplistic Lisp implementations, this list structure is directly interpreted to run the program; a function is literally a piece of list structure which is traversed by the interpreter in executing it. However, most actual Lisp systems (including all conforming Common Lisp systems) also include a compiler. The compiler translates list structure into machine code or bytecode for execution.

#### Evaluation and the read-eval-print loop

Lisp languages are frequently used with an interactive command line, which may be combined with an integrated development environment. The user types in expressions at the command line, or directs the IDE to transmit them to the Lisp system. Lisp *reads* the entered expressions, *evaluates* them, and *prints* the result. For this reason, the Lisp command line is called a "read-eval-print loop", or *REPL*.



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

The basic operation of the REPL is as follows. This is a simplistic description which omits many elements of a real Lisp, such as quoting and macros.

The read function accepts textual S-expressions as input, and parses them into an internal data structure. For instance, if you type the text (+ 1 2) at the prompt, read translates this into a linked list with three elements: the symbol +, the number 1, and the number 2. It so happens that this list is also a valid piece of Lisp code; that is, it can be evaluated. This is because the car of the list names a function—the addition operation.

Note that a foo will be read as a single symbol. 123 will be read as the number 123. "123" will be read as the string "123".

The eval function evaluates the data, returning zero or more other Lisp data as a result. Evaluation does not have to mean interpretation; some Lisp systems compile every expression to native machine code. It is simple, however, to describe evaluation as interpretation: To evaluate a list whose car names a function, eval first evaluates each of the arguments given in its cdr, then applies the function to the arguments. In this case, the function is addition, and applying it to the argument list (1 2) yields the answer 3. This is the result of the evaluation.

The symbol foo evaluates to the value of the symbol foo. Data like the string "123" evaluates to the same string. The list (quote (1 2 3)) evaluates to the list (1 2 3).

It is the job of the print function to represent output to the user. For a simple result such as 3 this is trivial. An expression which evaluated to a piece of list structure would require that print traverse the list and print it out as an S-expression.

To implement a Lisp REPL, it is necessary only to implement these three functions and an infinite-loop function. (Naturally, the implementation of eval will be complicated, since it must also implement all special operators like if or lambda.) This done, a basic REPL itself is but a single line of code: (loop (print (eval (read))))).

The Lisp REPL typically also provides input editing, an input history, error handling and an interface to the debugger.

Lisp is usually evaluated eagerly. In Common Lisp, arguments are evaluated in applicative order ('leftmost innermost'), while in Scheme order of arguments is undefined, leaving room for optimization by a compiler.

#### Control structures

Lisp originally had very few control structures, but many more were added during the language's evolution. (Lisp's original conditional operator, cond, is the precursor to later if-then-else structures.)

Programmers in the Scheme dialect often express loops using tail recursion. Scheme's commonality in academic computer science has led some students to believe that tail recursion is the only, or the most common, way to write iterations in Lisp, but this is incorrect. All frequently seen Lisp dialects have imperative-style iteration constructs, from Scheme's do loop to Common Lisp's complex loop expressions. Moreover, the key issue that makes this an objective rather than subjective matter is that Scheme makes specific requirements for the handling of tail calls, and consequently the reason that the use of tail recursion is generally encouraged for Scheme is that the practice is expressly supported by the language definition itself. By contrast, ANSI Common Lisp does not require the optimization commonly referred to as tail call elimination. Consequently, the fact that tail recursive style as a casual replacement for the use of more traditional iteration constructs (such as do, dolist or loop) is discouraged in Common Lisp is not just a matter of stylistic preference, but potentially one of efficiency (since an apparent tail call in Common Lisp may not compile as a simple jump) and program correctness (since tail recursion may increase stack use in Common Lisp, risking stack overflow).

Some Lisp control structures are *special operators*, equivalent to other languages' syntactic keywords. Expressions using these operators have the same surface appearance as function calls, but differ in that



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

the arguments are not necessarily evaluated or, in the case of an iteration expression, may be evaluated more than once.

In contrast to most other major programming languages, Lisp allows the programmer to implement control structures using the language itself. Several control structures are implemented as Lisp macros, and can even be macro-expanded by the programmer who wants to know how they work.

Both Common Lisp and Scheme have operators for non-local control flow. The differences in these operators are some of the deepest differences between the two dialects. Scheme supports *re-entrant continuations* using the call/cc procedure, which allows a program to save (and later restore) a particular place in execution. Common Lisp does not support re-entrant continuations, but does support several ways of handling escape continuations.

Frequently, the same algorithm can be expressed in Lisp in either an imperative or a functional style. As noted above, Scheme tends to favor the functional style, using tail recursion and continuations to express control flow. However, imperative style is still quite possible. The style preferred by many Common Lisp programmers may seem more familiar to programmers used to structured languages such as C, while that preferred by Schemers more closely resembles pure-functional languages such as Haskell.

Because of Lisp's early heritage in list processing, it has a wide array of higher-order functions relating to iteration over sequences. In many cases where an explicit loop would be needed in other languages (like a for loop in C) in Lisp the same task can be accomplished with a higher-order function. (The same is true of many functional programming languages.)

A good example is a function which in Scheme is called map and in Common Lisp is called mapcar. Given a function and one or more lists, mapcar applies the function successively to the lists' elements in order, collecting the results in a new list:

```
(mapcar #'(1 2 3 4 5) '(10 20 30 40 50))
```

This applies the + function to each corresponding pair of list elements, yielding the result (11 22 33 44 55).

Examples

Here are examples of Common Lisp code.

The basic "Hello world" program:

```
(print "Hello world")
```

As the reader may have noticed from the above discussion, Lisp syntax lends itself naturally to recursion. Mathematical problems such as the enumeration of recursively defined sets are simple to express in this notation.

Evaluate a number's factorial:

```
(defun factorial (n)
  (if (<= n 1)
      1
      (* n (factorial (- n 1)))))
```

An alternative implementation, often faster than the previous version if the Lisp system has tail recursion optimization:

```
(defun factorial (n &optional (acc 1))
  (if (<= n 1)
      acc
      (factorial (- n 1) (* acc n))))
```

Contrast with an iterative version which uses Common Lisp's loop macro:

```
(defun factorial (n)
  (loop for i from 1 to n
        for fac = 1 then (* fac i)
```



तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
Institute of Management & Technology  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

finally (return fac)))

The following function reverses a list. (Lisp's built-in *reverse* function does the same thing.)

```
(defun -reverse (list)
```

```
  (let ((return-value '()))
```

```
    (dolist (e list) (push e return-value))
```

```
    return-value))
```

List of programming languages for artificial intelligence

Artificial intelligence researchers have developed several specialized programming languages for artificial intelligence:**Languages**

- IPL was the first language developed for artificial intelligence. It includes features intended to support programs that could perform general problem solving, including lists, associations, schemas (frames), dynamic memory allocation, data types, recursion, associative retrieval, functions as arguments, generators (streams), and cooperative multitasking.
- Lisp is a practical mathematical notation for computer programs based on lambda calculus. Linked lists are one of Lisp languages' major data structures, and Lisp source code is itself made up of lists. As a result, Lisp programs can manipulate source code as a data structure, giving rise to the macro systems that allow programmers to create new syntax or even new domain-specific programming languages embedded in Lisp. There are many dialects of Lisp in use today; among them are Common Lisp, Scheme, and Clojure.
- Prolog is a declarative language where programs are expressed in terms of relations, and execution occurs by running *queries* over these relations. Prolog is particularly useful for symbolic reasoning, database and language parsing applications. Prolog is widely used in AI today.
- STRIPS is a language for expressing automated planning problem instances. It expresses an initial state, the goal states, and a set of actions. For each action preconditions (what must be established before the action is performed) and post conditions (what is established after the action is performed) are specified.
- Planner is a hybrid between procedural and logical languages. It gives a procedural interpretation to logical sentences where implications are interpreted with pattern-directed inference.

AI applications are also often written in standard languages like C++ and languages designed for mathematics, such as MATLAB and Lush.

#### Book References:-

- E. Rich and K. Knight, "Artificial intelligence", TMH, 2nd ed., 1999.
- D.W. Patterson, "Introduction to AI and Expert Systems", PHI, 1999
- Nils J Nilsson, "Artificial Intelligence -A new Synthesis" 2nd Edition (2000), Harcourt Asia Ltd.

#### Web References:-

- Aamodt, A. and Plaza, E. (1994). Case-based reasoning: foundational issues, methodological variations, and system approaches. AI Communications, 7(1): 39-59.
- Abelson, H. and DiSessa, A. (1981). Turtle Geometry: The Computer as a Medium for Exploring Mathematics. MIT Press, Cambridge, MA.





तेजस्वि नावधीतमस्तु  
ISO 9001:2008 & 14001:2004

**FAIRFIELD**  
**Institute of Management & Technology**  
Managed by 'The Fairfield Foundation'  
( Affiliated to GGSIP University, New Delhi )

- Abramson, H. and Rogers, M.H. (Eds.) (1989). Meta-Programming in Logic Programming. MIT Press, Cambridge, MA.
- Agre, P.E. (1995). Computational research on interaction and agency. Artificial Intelligence, 72: 1-52.
- Albus, J.S. (1981). Brains, Behavior and Robotics. BYTE Publications, Peterborough, NH.
- Allais, M. and Hagen, O. (Eds.) (1979). Expected Utility Hypothesis and the Allais Paradox. Reidel, Boston, MA.
- Allen, J., Hendler, J., and Tate, A. (Eds.) (1990). Readings in Planning. Morgan Kaufmann, San Mateo, CA.
- Anderson, M. and Leigh Anderson, S.L. (2007). Machine ethics: Creating an ethical intelligent agent. AI Magazine, 28(4): 15-26.
- Andrieu, C., de Freitas, N., Doucet, A., and Jordan, M.I. (2003). An introduction to MCMC for machine learning. Machine Learning, 50(1-2): 5-43.
- Antoniou, G. and van Harmelen, F. (2008). A Semantic Web Primer. MIT Press, Cambridge, MA, 2nd edition.