Chapter 1
Introduction to Artificial Intelligence

1. Introduction

A branch of Computer Science named Artificial Intelligence (AI) pursues creating the computers / machines as intelligent as human beings. John McCarthy the father of Artificial Intelligence described AI as, *"The science and engineering of making intelligent machines, especially intelligent computer programs".* **A**rtificial Intelligence (AI) is a branch of *Science* which deals with helping machines find solutions to complex problems in a more human-like fashion.

- This generally involves borrowing characteristics from human intelligence, and applying them as algorithms in a computer friendly way.
- A more or less flexible or efficient approach can be taken depending on the requirements established, which influences how artificial the intelligent behavior appears.

Artificial is defined in different approaches by various researchers during its evolution, such as "Artificial Intelligence is the study of how to make computers do things which at the moment, people do better."
There are other possible definitions "**AI is a collection of hard problems which can be solved by humans and other living things, but for which we don't have good algorithms for solving."**

e. g., understanding spoken natural language, medical diagnosis, circuit design, learning, self-adaptation, reasoning, chess playing, proving math theories, etc.

1.1AI Approaches

The difference between machine and human intelligence is that the human think / act rationally compare to machine. Historically, all four approaches to AI have been followed, each by different people with different methods.

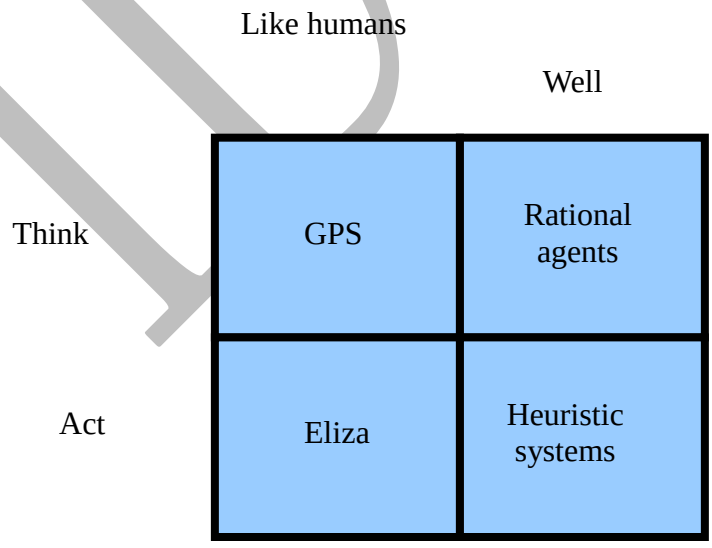


Figure 1: AI approaches

Think well

Develop formal models of knowledge representation, reasoning, learning, memory, problem solving that can be rendered in algorithms.
There is often an emphasis on a systems that are provably correct, and guarantee finding an optimal solution.

Act well

- For a given set of inputs, generate an appropriate output that is not necessarily correct but gets the job done.
- A **heuristic** (**heuristic rule**, **heuristic method**) is a rule of thumb, strategy, trick, simplification, or any other kind of device which drastically limits search for solutions in large problem spaces.

Heuristics do not guarantee optimal solutions; in fact, they do not guarantee any solution at all: **all that can be said for a useful heuristic is that it offers solutions which are good enough most of the time**

Think like humans

Cognitive science approach. Focus not just on behavior and I/O but also look at reasoning process. The Computational model should reflect "how" results were obtained. Provide a new language for expressing cognitive theories and new mechanisms for evaluating them. GPS (General Problem Solver): Goal not just to produce humanlike behavior (like ELIZA), but to produce a sequence of steps of the reasoning process that was similar to the steps followed by a person in solving the same task.

Act like humans

Behaviorist approach- Not interested in how you get results, just the similarity to what human results are. Exemplified by the Turing Test (Alan Turing, 1950).

Example:
ELIZA: A program that simulated a psychotherapist interacting with a patient and successfully passed the Turing Test. It was coded at MIT during 1964-1966 by Joel Weizenbaum. First script was DOCTOR. The script was a simple collection of syntactic patterns not unlike regular expressions Each pattern had an associated reply which might include bits of the input (after simple transformations (my → your) Weizenbaum was shocked at reactions: Psychiatrists thought it had potential. People unequivocally anthropomorphized. Many thought it solved the NL problem.

1.2 Applications of Artificial Intelligence

In 1981 In Artificial Intelligence became an Industry means the theories and techniques proposed in Artificial Intelligence literatures were accepted by Industries and they started investing in the area of Artificial Intelligence.

The concepts of Artificial Intelligence are implemented in various fields. Few of them are listed as follows.

- **Game Playing**

AI plays crucial role in strategic games such as chess, poker, tic-tac-toe, etc., where machine can think of large number of possible positions based on heuristic knowledge.

- **Speech Recognition**
  Some intelligent systems are capable of hearing and comprehending the language in terms of sentences and their meanings while a human talks to it. It can handle different accents, slang words, noise in the background, change in human's noise due to cold, etc.

- **Understanding Natural Language**
  The computer can now understand natural languages and hence human can now interact using natural spoken languages. The computer has to be provided with an understanding of the domain the text is about, and this is presently possible only for very limited domains.

- **Computer Vision**

  The computer vision leads the computer to understand the signals and act accordingly. There are some systems as:
  - Face detection system installed at airport.
  - Medical diagnosis

- **Expert Systems**
  A ``knowledge engineer'' interviews experts in a certain domain and tries to embody their knowledge in a computer program for carrying out some task. How well this works depends on whether the intellectual mechanisms required for the task are within the present state of AI. When this turned out not to be so, there were many disappointing results. One of the first expert systems was MYCIN in 1974, which diagnosed bacterial infections of the blood and suggested treatments. It did better than medical students or practicing doctors, provided its limitations were observed. Namely, its ontology included bacteria, symptoms, and treatments and did not include patients, doctors, hospitals, death, recovery, and events occurring in time. Its interactions depended on a single patient being considered. Since the experts consulted by the knowledge engineers knew about patients, doctors, death, recovery, etc., it is clear that the knowledge engineers forced what the experts told them into a predetermined framework. The usefulness of current expert systems depends on their users having common sense.

- **Heuristic Classification**
  One of the most feasible kinds of expert system given the present knowledge of AI is to put some information in one of a fixed set of categories using several sources of information. An example is advising whether to accept a proposed credit card purchase. Information is available about the owner of the credit card, his record of payment and also about the item he is buying and about the establishment from which he is buying it.

- **Intelligent Robots**
  Robots have sensors to detect physical data from the real world such as light, heat, temperature, movement, sound, bump, and pressure. They have efficient processors, multiple sensors and huge memory, to exhibit intelligence.

Apart from regular application there are some domain task covered by Artificial Intelligence which are listed as follows:

Perception

- Vision
- Speech

Natural Language

- Understanding
- Generation
- Translation

Common Sense Reasoning

Robot Control

Games

- Chess
- Backgammon
- Checkers
- Go

Mathematics

- Geometry
- Logic
- Integral Calculus

Expert Tasks

- Design
- Fault finding
- Manufacturing Planning

Scientific Analysis

Medical Diagnosis

Financial analysis

1.3 Historical Evolution of Artificial Intelligence

Intelligence forms the foundation of all human technology and in fact all human civilization. But there was a feeling that the human efforts to learn knowledge constitute a transgression against the laws of God or nature. E.g. Eve in Bible, Prometheus in Greek mythology. The logical starting point of the history in AI dates back to Aristotle. He formalized the insights, wonders and fears regarding nature with careful analysis into a disciplined thought. For him, the study of thought it self was the basis of all knowledge. In his 'Logic', he investigated whether certain propositions can be said to be "true" because they are related to other things that are known to be true. Gottlob Frege, Bertrand Russel, Kurt Godel, Alan Turing etc followed this school of thought. The major development, which drastically changed the world view was the discovery of Copper Niccus - that the earth is not the center of universe but is a just a component of it. Though it was against the practiced dogmas and revered religious beliefs, it was a new realization that -

- Our ideas about world may be fundamentally different from its appearance.
- There is a gap between the human mind and its surrounding realities.
- There is a gap between idea about things and things about themselves.

The argument is that-

- We could separate mind and physical world.

- It is necessary to find way to connect these.

The accepted view is that, though they are separate, they are not fundamentally different. Mental process, like physical process, can be characterized using formal mathematics or logic.

The historical events are summarize in the following table:

Year	Event
1923	Karel Čapek's play named "Rossum's Universal Robots" (RUR) opens in London, first use of the word "robot" in English.
1943	Foundations for neural networks laid.
1945	Isaac Asimov, a Columbia University alumni, coined the term *Robotics.*
1950	Alan Turing introduced Turing Test for evaluation of intelligence and published *Computing Machinery and Intelligence.* Claude Shannon published *Detailed Analysis of Chess Playing* as a search.
1956	John McCarthy coined the term *Artificial Intelligence.* Demonstration of the first running AI program at Carnegie Mellon University.
1958	John McCarthy invents LISP programming language for AI.
1964	Danny Bobrow's dissertation at MIT showed that computers can understand natural language well enough to solve algebra word problems correctly.
1965	Joseph Weizenbaum at MIT built *ELIZA*, an interactive problem that carries on a dialogue in English.
1969	Scientists at Stanford Research Institute Developed *Shakey*, a robot, equipped with locomotion, perception, and problem solving.
1973	The Assembly Robotics group at Edinburgh University built *Freddy*, the Famous Scottish Robot, capable of using vision to locate and assemble models.
1979	The first computer-controlled autonomous vehicle, Stanford Cart, was built.
1985	Harold Cohen created and demonstrated the drawing program, *Aaron*.
1990	Major advances in all areas of AI: • Significant demonstrations in machine learning • Case-based reasoning • Multi-agent planning • Scheduling • Data mining, Web Crawler • natural language understanding and translation • Vision, Virtual Reality • Games
1997	The Deep Blue Chess Program beats the then world chess champion, Garry Kasparov.
2000	Interactive robot pets become commercially available. MIT displays *Kismet*, a robot with a face that expresses emotions. The robot *Nomad* explores remote regions of Antarctica and locates meteorites.

1.4Intelligent System

The intelligent system part of artificial intelligence is defined as, the system that the ability to calculate, reasoning, perception, store and retrieve the information from memory, solving complex problems and adaption.

1.4.1 Types of Intelligence

Intelligence	Description	Example
Linguistic intelligence	The ability to speak, recognize, and use mechanisms of phonology (speech sounds), syntax (grammar), and semantics (meaning).	Narrators, Orators
Musical intelligence	The ability to create, communicate with, and understand meanings made of sound, understanding of pitch, rhythm.	Musicians, Singers, Composers
Logical-mathematical intelligence	The ability of use and understand relationships in the absence of action or objects. Understanding complex and abstract ideas.	Mathematicians, Scientists
Spatial intelligence	The ability to perceive visual or spatial information, change it, and re-create visual images without reference to the objects, construct 3D images, and to move and rotate them.	Map readers, Astronauts, Physicists
Bodily-Kinesthetic intelligence	The ability to use complete or part of the body to solve problems or fashion products, control over fine and coarse motor skills, and manipulate the objects.	Players, Dancers
Intra-personal intelligence	The ability to distinguish among one's own feelings, intentions, and motivations.	Gautam Buddha
Interpersonal intelligence	The ability to recognize and make distinctions among other people's feelings, beliefs, and intentions.	Mass Communicators, Interviewers.

Intelligent System, is Composed of :

The intelligence is intangible. It is composed of −

- Reasoning
- Learning
- Problem Solving
- Perception
- Linguistic Intelligence

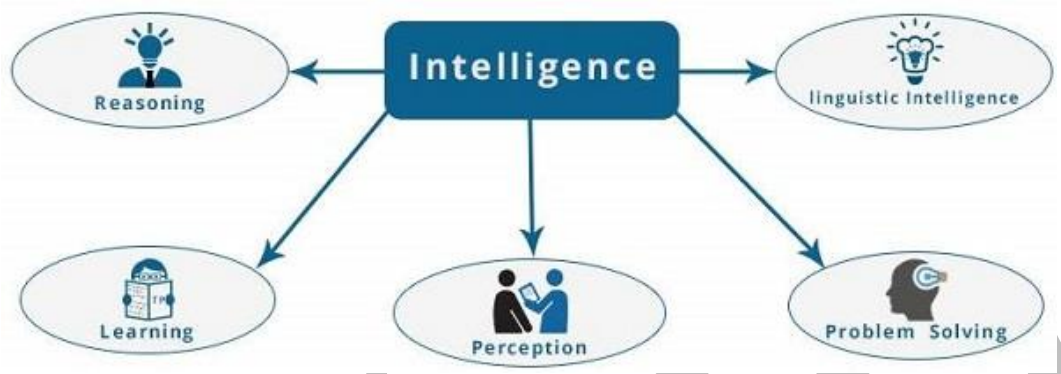


Figure 1.1: Composition of Artificial Intelligence

Let us go through all the components briefly −

- **Reasoning** − It is the set of processes that enables us to provide basis for judgement, making decisions, and prediction. There are broadly two types −

Inductive Reasoning	Deductive Reasoning
It conducts specific observations to makes broad general statements.	It starts with a general statement and examines the possibilities to reach a specific, logical conclusion.
Even if all of the premises are true in a statement, inductive reasoning allows for the conclusion to be false.	If something is true of a class of things in general, it is also true for all members of that class.
Example − "Tarush is a teacher. Tarush is studious. Therefore, All teachers are studious."	Example − "All women of age above 60 years are grandmothers. Vijaya is 65 years. Therefore, Vijaya is a grandmother."

- **Learning** − It is the activity of gaining knowledge or skill by studying, practicing, being taught, or experiencing something. Learning enhances the awareness of the subjects of the study.

  The ability of learning is possessed by humans, some animals, and AI-enabled systems. Learning is categorized as −

  - **Auditory Learning** − It is learning by listening and hearing. For example, students listening to recorded audio lectures.
  - **Episodic Learning** − To learn by remembering sequences of events that one has witnessed or experienced. This is linear and orderly.

- **Motor Learning** − It is learning by precise movement of muscles. For example, picking objects, Writing, etc.
- **Observational Learning** − To learn by watching and imitating others. For example, child tries to learn by mimicking her parent.
- **Perceptual Learning** − It is learning to recognize stimuli that one has seen before. For example, identifying and classifying objects and situations.
- **Relational Learning** − It involves learning to differentiate among various stimuli on the basis of relational properties, rather than absolute properties. For Example, Adding 'little less' salt at the time of cooking potatoes that came up salty last time, when cooked with adding say a tablespoon of salt.
- **Spatial Learning** − It is learning through visual stimuli such as images, colors, maps, etc. For Example, A person can create roadmap in mind before actually following the road.
- **Stimulus-Response Learning** − It is learning to perform a particular behavior when a certain stimulus is present. For example, a dog raises its ear on hearing doorbell.

- **Problem Solving** − It is the process in which one perceives and tries to arrive at a desired solution from a present situation by taking some path, which is blocked by known or unknown hurdles.

  Problem solving also includes **decision making**, which is the process of selecting the best suitable alternative out of multiple alternatives to reach the desired goal are available.

- **Perception** − It is the process of acquiring, interpreting, selecting, and organizing sensory information. Perception presumes **sensing**. In humans, perception is aided by sensory organs. In the domain of AI, perception mechanism puts the data acquired by the sensors together in a meaningful manner.
- **Linguistic Intelligence** − It is one's ability to use, comprehend, speak, and write the verbal and written language. It is important in interpersonal communication.

1.4.2 Difference between Human and Machine Intelligence

- Humans perceive by patterns whereas the machines perceive by set of rules and data.
- Humans store and recall information by patterns, machines do it by searching algorithms. For example, the number 40404040 is easy to remember, store, and recall as its pattern is simple.
- Humans can figure out the complete object even if some part of it is missing or distorted; whereas the machines cannot do it correctly.

1.5 Agent and Environment

An **agent** is anything that perceive its environment through **sensors** and acts upon that environment through **effectors.**

- A **human agent** has sensory organs such as eyes, ears, nose, tongue and skin parallel to the sensors, and other organs such as hands, legs, mouth, for effectors.
- A **robotic agent** replaces cameras and infrared range finders for the sensors, and various motors and actuators for effectors.
- A **software agent** has encoded bit strings as its programs and actions.

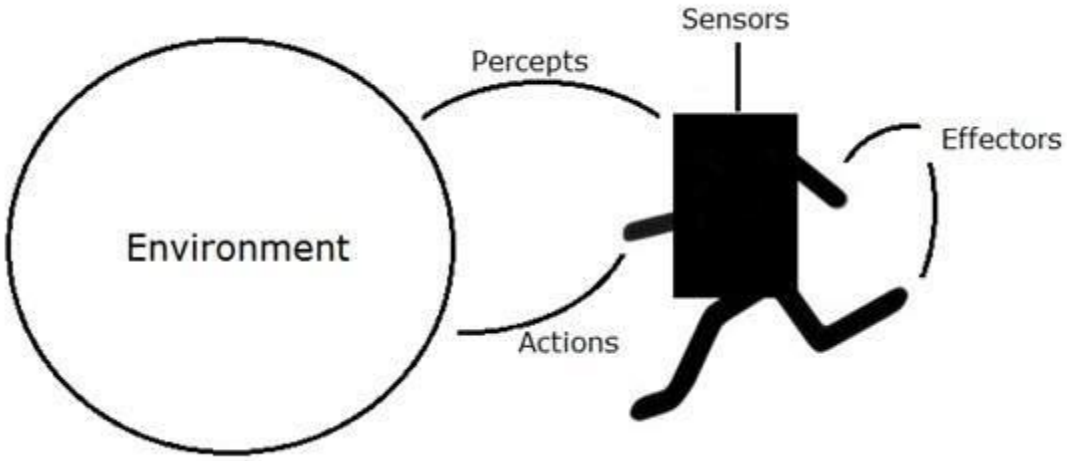


Figure 1.2: Agent and Environment

1.5.1 Agent Terminology

- **Performance Measure of Agent** − It is the criteria, which determines how successful an agent is.
- **Behavior of Agent** − It is the action that agent performs after any given sequence of percepts.
- **Percept** − It is agent's perceptual inputs at a given instance.
- **Percept Sequence** − It is the history of all that an agent has perceived till date.
- **Agent Function** − It is a map from the precept sequence to an action.

Rationality

Rationality is nothing but status of being reasonable, sensible, and having good sense of judgment.

Rationality is concerned with expected actions and results depending upon what the agent has perceived. Performing actions with the aim of obtaining useful information is an important part of rationality.

1.5.2 Ideal Rational Agent

An ideal rational agent is the one, which is capable of doing expected actions to maximize its performance measure, on the basis of −

- Its percept sequence
- Its built-in knowledge base

Rationality of an agent depends on the following −

- The **performance measures**, which determine the degree of success.
- Agent's **Percept Sequence** till now.
- The agent's **prior knowledge about the environment**.
- The **actions** that the agent can carry out.

A rational agent always performs right action, where the right action means the action that causes the agent to be most successful in the given percept sequence. The problem the agent solves is characterized by Performance Measure, Environment, Actuators, and Sensors (PEAS).

1.5.3 The Structure of Intelligent Agents

Agent's structure can be viewed as −

- Agent = Architecture + Agent Program
- Architecture = the machinery that an agent executes on.
- Agent Program = an implementation of an agent function.

(a) Simple Reflex Agents

- They choose actions only based on the current percept.
- They are rational only if a correct decision is made only on the basis of current precept.
- Their environment is completely observable.

Condition-Action Rule − It is a rule that maps a state (condition) to an action.

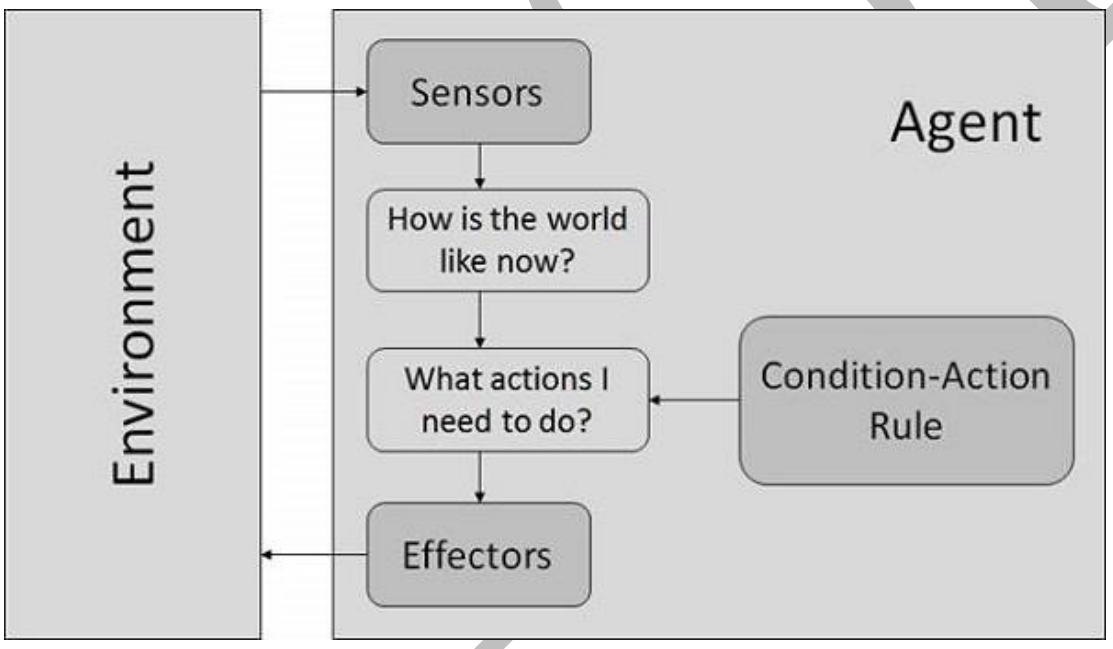


Figure 1.3a: Agents and Environment

(b) Model Based Reflex Agents

They use a model of the world to choose their actions. They maintain an internal state.

Model − knowledge about "how the things happen in the world".

Internal State − It is a representation of unobserved aspects of current state depending on percept history.

Updating the state requires the information about −

- How the world evolves.
- How the agent's actions affect the world.

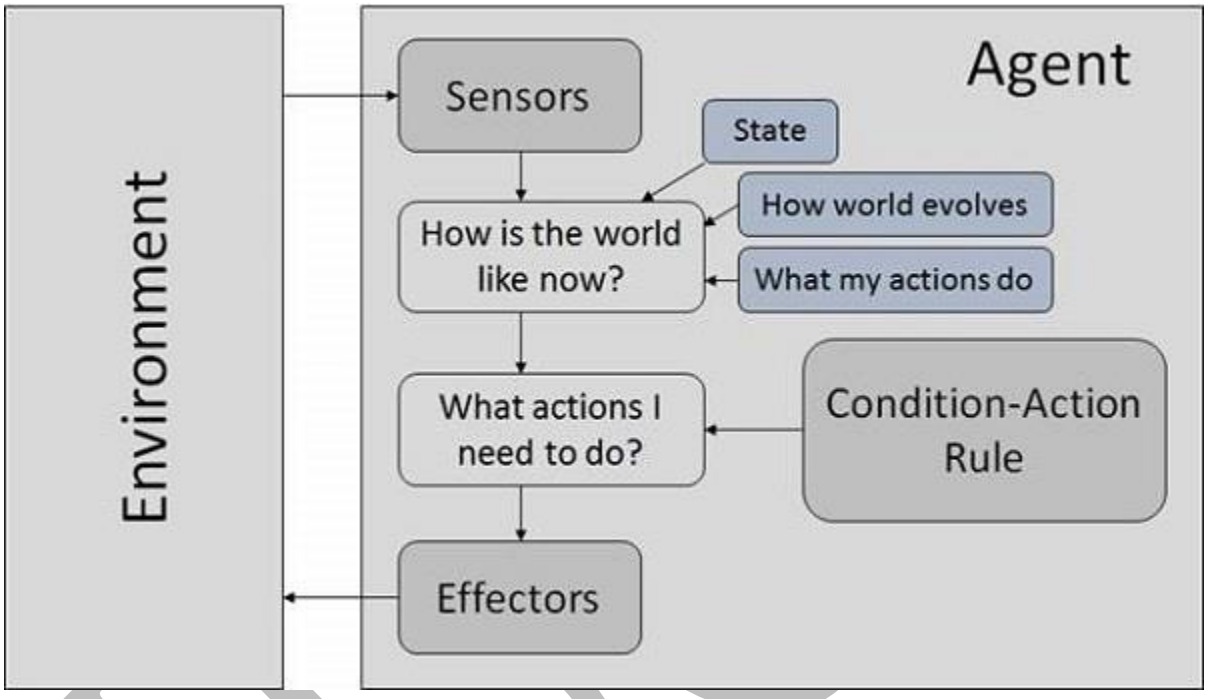


Figure 1.3 b: Agents and Environment

(c) Goal Based Agents

They choose their actions in order to achieve goals. Goal-based approach is more flexible than reflex agent since the knowledge supporting a decision is explicitly modeled, thereby allowing for modifications.

Goal − It is the description of desirable situations.

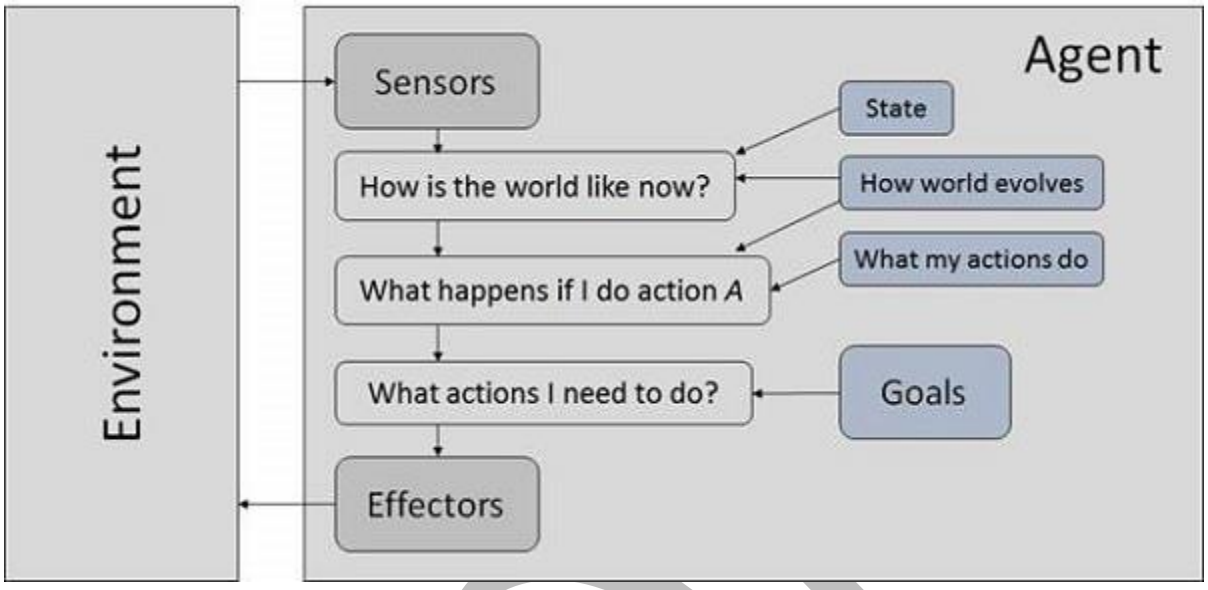


Figure 1.3c: Agents and Environment

(d) Utility Based Agents

They choose actions based on a preference (utility) for each state.

Goals are inadequate when −

- There are conflicting goals, out of which only few can be achieved.
- Goals have some uncertainty of being achieved and you need to weigh likelihood of success against the importance of a goal.

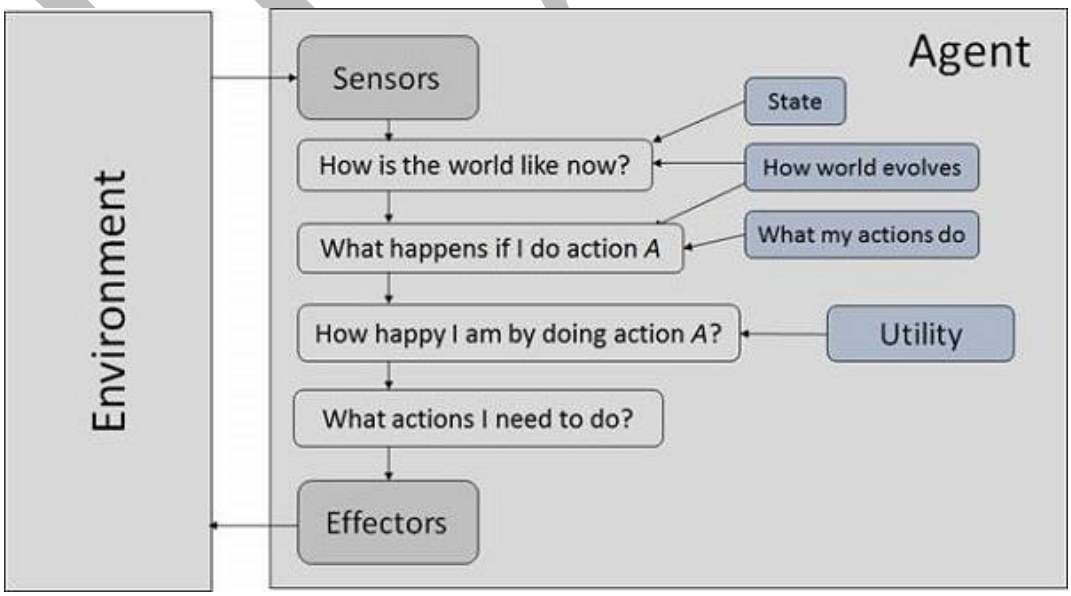


Figure 1.3c: Agents and Environment

e) Turing Test

The success of an intelligent behavior of a system can be measured with Turing Test.

Two persons and a machine to be evaluated participate in the test. Out of the two persons, one plays the role of the tester. Each of them sits in different rooms. The tester is unaware of who is machine and who is a human. He interrogates the questions by typing and sending them to both intelligences, to which he receives typed responses.

TURING TEST

Allan Turing, in 1950, considered the question of whether a machine could actually make to think. Turing test measures the performance of an 'intelligent' machine.

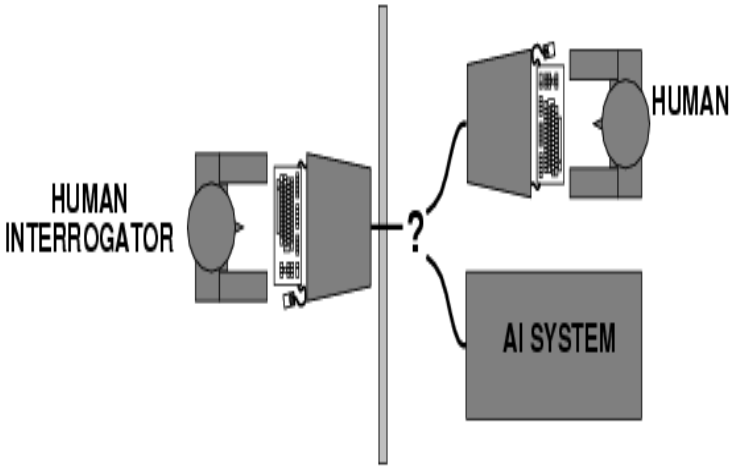


Figure 1.4: Turing Test

This test aims at fooling the tester. If the tester fails to determine machine's response from the human response, then the machine is said to be intelligent.

1.5.4 The Nature of Environments

Some programs operate in the entirely **artificial environment** confined to keyboard input, database, computer file systems and character output on a screen.

In contrast, some software agents (software robots or softbots) exist in rich, unlimited softbots domains. The simulator has a **very detailed, complex environment**. The software agent needs to choose from a long array of actions in real time. A softbot designed to scan the online preferences of the customer and show interesting items to the customer works in the **real** as well as an **artificial** environment.

The most famous **artificial environment** is the **Turing Test environment**, in which one real and other artificial agents are tested on equal ground. This is a very challenging environment as it is highly difficult for a software agent to perform as well as a human.

1.5 Properties of Environment

The environment has multifold properties −

- **Discrete / Continuous** − If there are a limited number of distinct, clearly defined, states of the environment, the environment is discrete (For example, chess); otherwise it is continuous (For example, driving).
- **Observable / Partially Observable** − If it is possible to determine the complete state of the environment at each time point from the percepts it is observable; otherwise it is only partially observable.
- **Static / Dynamic** − If the environment does not change while an agent is acting, then it is static; otherwise it is dynamic.
- **Single agent / Multiple agents** − The environment may contain other agents which may be of the same or different kind as that of the agent.
- **Accessible / Inaccessible** − If the agent's sensory apparatus can have access to the complete state of the environment, then the environment is accessible to that agent.
- **Deterministic / Non-deterministic** − If the next state of the environment is completely determined by the current state and the actions of the agent, then the environment is deterministic; otherwise it is non-deterministic.
- **Episodic / Non-episodic** − In an episodic environment, each episode consists of the agent perceiving and then acting. The quality of its action depends just on the episode itself. Subsequent episodes do not depend on the actions in the previous episodes. Episodic environments are much simpler because the agent does not need to think ahead.

1.6 Logic Based Intelligence

- As thinking has become a form of computation, its formalization and mechanization became necessary.
- In 1887, Leibniz introduced a system of formal logic and constructed a machine for automating its calculation.
- Euler's discovery of graph theory, through the Kongsberg problem, introduced the concept of state space representation.
- George Boole, through his Boolean algebra, made the mathematical formalization of the laws of logic through AND, OR & NOT.
- Gottlob Frege created a language, called 'First Order Predicate Calculus', which later used as a representation techniques in AI.
- Bertrand Russel, in his 'Principia Mathematica', developed theoretical foundation of AI by treating mathematics as a formal system.
- Alfred Tarski created a 'Theory of Reference' wherein the well-formed formulae can be referred.
- Though study of science and mathematics were the prerequisites for the formal study of AI, by the invention of digital computers it was changed.

- Using its potential to provide memory and processing power, we can treat Intelligence as a form information processing.
- Search as a problem solving methodology.
- Knowledge can be put in a representational way, which can be manipulated using algorithms.

THINKING MACHINES

- Can machines think?
- Are human's machines?
- Can machines think like humans?
- How would we know whether a computer is thinking?

CAN COMPUTERS THINK?

- Some common answers:
  No (dualist/mystic): Computers lack "mental stuff". They don't have intuitions, feelings, phenomenology.(Soul?)
  No (neurophysiology critical): Even if their behavior is arbitrary close, biology is essential.
  No (beyond our capabilities): Not impossible in practice, but too complex. There are limits on our self-knowledge which will prevent us from creating a thinking computer.
  Yes (but not in our lifetimes): Too complex. Practical obstacles may be insurmountable. But with better science and technology, maybe.
  Yes (functionalist): Programs/computers will become smarter without a clear limit. For all practical purposes, they will 'think' because they will perform the FUNCTIONS of thinking. (The standard AI answer.)
  Yes (extreme functionalist; back to mystic): Computers already think.
- All matter exhibits mind in different aspects and degrees.

1.7 Logic Based Intelligence Vs. Agent Based Intelligence

- We have looked at intelligence as a logical inference and logic as a knowledge representation technique.
- This is a typical western thought starting from Aristotle.
- Of late, there have been instances of questioning this school of thought.
- This argument is based on simple facts like –
  - Formation of Language has no logical reasoning. It has much to do with cultural and social situation.
  - Working of brain is based on the inputs of neurons (So we have Artificial Neural Network).
  - Specie's adapt to an environment (So we have Genetic Algorithm).
  - Working of social systems is based on the performance of autonomous individual agents (So we have Intelligent Agents).
- These examples show two things -
  - Intelligence is emerged in the process.
  - Intelligence is reflected by the collective behavior of agents.

- Agent-oriented and emergent view of intelligence has the following properties –
  - Agents are autonomous or semi-autonomous. Each agent has a specified responsibility to undertake and is ignorant about what others are doing.
  - Each agent is sensitive to its own surrounding environment and has no knowledge of the full domain.
  - Agents interact one another. The society of agents is structured, helping in the solving of global problem.
  - The cooperative interaction of the agents results in the emergence of intelligence.

Questions:

1. What is Artificial Intelligence? State its applications.
2. Discuss the historical evolution of Artificial Intelligence.
3. State the relationship between agent and environment.
4. What is agent? State its types.
5. Explain properties of environment.
6. Explain different types of Intelligent System.
7. Write a note on Turing Test.
8. Give comparison between logic based intelligence and agent based intelligence.

Chapter 2
AI Representation and Predicates

2.1 Introduction:

Knowledge is a general term. To represent knowledge it requires an analysis to distinguish between knowledge "how" and knowledge "that".

- knowing "how to do something". e.g. "how to drive a car" is Procedural knowledge.
- knowing "that something is true or false". e.g. "that is the speed limit for a car on a motorway" is Declarative knowledge.

Knowledge and Representation are distinct entities that play a central but distinguishable roles in intelligent system.

- Knowledge is a description of the world. It determines a system's competence by what it knows.
- Representation is the way knowledge is encoded. It defines the performance of a system in doing something. • Different types of knowledge require different kinds of representation. The Knowledge Representation models/mechanisms are often based on: ◊ Logic ◊ Rules ◊ Frames ◊ Semantic Net.

Knowledge representation and reasoning is establishing a relationship between human knowledge and its representation, using formal languages, within the computer.

There are different fields that contributed to AI in the form of ideas, viewpoints and techniques given as follows:

Philosophy: Logic, reasoning, mind as a physical system, foundations of learning, language and rationality.
Mathematics: Formal representation and proof algorithms, computation, (un) decidability, (in) tractability, probability.
Psychology: adaptation, phenomena of perception and motor control.
Economics: formal theory of rational decisions, game theory.
Linguistics: knowledge representation, grammar.
Neuroscience: physical substrate for mental activities.
Control theory: homeostatic systems, stability, and optimal agent design.

2.2 A Brief History

After world war II in 1943, Warren Mc Culloch and Walter Pitts proposed a model of artificial Boolean neurons to perform computations. The first step toward connectionist computation and learning known as (Hebbian learning). Marvin Minsky and Dann Edmonds (1951) constructed the first neural network computer.

1950: Alan Turing's "Computing Machinery and Intelligence" was the first complete vision of AI.

The birth of AI took place (1956) at Dartmouth conference, there was probably the first workshop of Artificial Intelligence and with it the field of AI research was born. Researcher from Carnegie Mellon University (CMU), Massachusetts Institute of Technology (MIT) and employee from IBM met together and founded the AI research. In the following years they made huge process. Nearly everybody was very optimistic.
"*Machines will be capable, within twenty years, of doing any work what man can do." – Herbert A. Simon (CMU)" "Within a generation ... the problem of creating 'artificer intelligence' will substantially be solved" – Marvin Minsky (MIT).* The progress slowed down in the following years. Because of the failing recognizing of the difficulty of the tasks promises were broken.

- Bertrand Russell and Alfred North Whitehead published *Principia Mathematica*, which revolutionaized formal logic. Russell, Ludwig Wittgenstein, and Rudolf Carnap lead philosophy into logical analysis of knowledge.
- Torres y Quevedo built his chess machine 'Ajedrecista', using electromagnets under the board to play the endgame rook and king against the lone king, possibly the first computer game (1912).
- Karel Capek's play *"R.U.R." (Rossum's Universal Robots)* produced in 1921 (London opening, 1923). - First use of the word 'robot' in English.
- Alan Turing proposed the universal Turing machine (1936-37)
- Electro, a mechanical man, introduced by Westinghouse Electricat the World's Fair in New York (1939), along with Sparko, a mechanical dog.
- Warren McCulloch & Walter Pitts publish "A Logical Calculus of the Ideas Immanent in Nervous Activity" (1943), laying foundations for neural networks.
- Arturo Rosenblueth, Norbert Wiener & Julian Bigelow coin the term "cybernetics" in a 1943 paper. Wiener's popular book by that name published in 1948.
- Emil Post proves that production systems are a general computational mechanism (1943). See Ch.2 of Rule Based Expert Systems for the uses of production systems in AI. Post also did important work on completeness, inconsistency, and proof theory.
- George Polya published his best-selling book on thinking heuristically, How to Solve It in 1945. This book introduced the term 'heuristic' into modern thinking and has influenced many AI scientists.
- Vannevar Bush published As We May Think (Atlantic Monthly, July 1945) a prescient vision of the future in which computers assist humans in many activities.
- Grey Walter experimented with autonomous robots, turtles named Elsie and Elmer, at Bristol (1948-49) based on the premise that a small number of brain cells could give rise to complex behaviors.
- A.M. Turing published "Computing Machinery and Intelligence" (1950). - Introduction of Turing Test as a way of operationalizing a test of intelligent behavior. See The Turing Institute for more on Turing.
- Claude Shannon published detailed analysis of chess playing as search in "Programming a computer to play chess" (1950).
- Isaac Asimov published his three laws of robotics (1950).

(a) Modern History

The modern history of AI begins with the development of stored-program electronic computers. For a short summary, see Genius and Tragedy at Dawn of Computer Age By ALICE RAWSTHORN, NY

Times (March 25, 2012), a review of technology historian George Dyson's book “Turing’s Cathedral: The Origins of the Digital Universe.”

1956

- John McCarthy coined the term "artificial intelligence" as the topic of the Dartmouth Conference, the first conference devoted to the subject.
- Demonstration of the first running AI program, the Logic Theorist (LT) written by Allen Newell, J.C. Shaw and Herbert Simon (Carnegie Institute of Technology, now Carnegie Mellon University). See Over the holidays 50 years ago, two scientists hatched artificial intelligence.

1957

- The General Problem Solver (GPS) demonstrated by Newell, Shaw & Simon.

1958 - 1975

- John McCarthy (MIT) invented the Lisp language.
- Herb Gelernter & Nathan Rochester (IBM) described a theorem prover in geometry that exploits a semantic model of the domain in the form of diagrams of "typical" cases.
- Teddington Conference on the Mechanization of Thought Processes was held in the UK and among the papers presented were John McCarthy's Programs with Common Sense, " Oliver Selfridge's "Pandemonium," and Marvin Minsky's "Some Methods of Heuristic Programming and Artificial Intelligence."

1975- 1985

- Mycin program, initially written as Ted Shortliffe's Ph.D. dissertation at Stanford, was demonstrated to perform at the level of experts. Bill VanMelle's PhD dissertation at Stanford demonstrated the generality of MYCIN's representation of knowledge and style of reasoning in his EMYCIN program, the model for many commercial expert system "shells".
- Jack Myers and Harry Pople at University of Pittsburgh developed INTERNIST, a knowledge-based medical diagnosis program based on Dr. Myers' clinical knowledge.
- Cordell Green, David Barstow, Elaine Kant and others at Stanford demonstrated the CHI system for automatic programming.
- The Stanford Cart, built by Hans Moravec, becomes the first computer-controlled, autonomous vehicle when it successfully traverses a chair-filled room and circumnavigates the Stanford AI Lab.
- Drew McDermott & Jon Doyle at MIT, and John McCarthy at Stanford begin publishing work on non-monotonic logics and formal aspects of truth maintenance.

1985- 1990

- Lisp Machines developed and marketed.
- First expert system shells and commercial applications.

2.3 State Of The Art

- Deep Blue defeated the reigning world chess champion Garry Kasparov in 1997.
- ALVINN: No hands across America (driving autonomously 98% of the time from Pittsburgh to San Diego).
- DART: During the 1991 Gulf War, US forces deployed an AI logistics planning and scheduling program that involved up to 50,000 vehicles, cargo, and people.
- NASA's on-board autonomous planning program controlled the scheduling of operations for a spacecraft.
- Proverb solves crossword puzzles better than most humans.

2.4 Knowledge representation requirements

In any intelligent system, representing the knowledge is supposed to be an important technique to encode the knowledge. The main objective of AI system is to design the programs that provide information to the computer, which can be helpful to interact with humans and solve problems in various fields which require human intelligence.

2.4.1 Knowledge can be defined as:

- an useful term to judge the understanding of an individual on a given subject.
- In intelligent systems, domain is the main focused subject area. So, the system specifically focuses on acquiring the domain knowledge.

2.4.2 Types of knowledge in AI

Based on the type of functionality, the knowledge in AI is categorized as:

i. Declarative knowledge

- The knowledge which is based on concepts, facts and objects, is termed as 'Declarative Knowledge'.
- It provides all the necessary information about the problem in terms of simple statements, either true or false.

ii. Procedural knowledge

- Procedural knowledge derives the information on the basis of rules, strategies, agendas and procedure.
- It describes how a problem can be solved.
- Procedural knowledge directs the steps on how to perform something.
  For example: Computer program.

iii. Heuristic knowledge

- Heuristic knowledge is based on thumb rule.

- It provides the information based on a thumb rule, which is useful in guiding the reasoning process.
- In this type, the knowledge representation is based on the strategies to solve the problems through the experience of past problems, compiled by an expert. Hence, it is also known as **Shallow knowledge.**

iv. Meta-knowledge

- This type gives an idea about the other types of knowledge that are suitable for solving problem.
- Meta-knowledge is helpful in enhancing the efficiency of problem solving through proper reasoning process.

v. Structural knowledge

- Structural knowledge is associated with the information based on rules, sets, concepts and relationships.
- It provides the information necessary for developing the knowledge structures and overall **mental model** of the problem.

2.4.3 Challenges in knowledge representation

The main objective of knowledge representation is to draw the conclusions from the knowledge, but there are many issues associated with the use of knowledge representation techniques.

For a knowledge-representation language to be practical, there are particular requirements that must be fulfilled:
• It must have a precise semantics (i.e., in terms of mathematical objects such as sets, numbers, etc.);
• There must be information available about the relationship between expressive power, i.e., what one can express in the language, and computational complexity in terms of required running time and space of the associated reasoning algorithms.

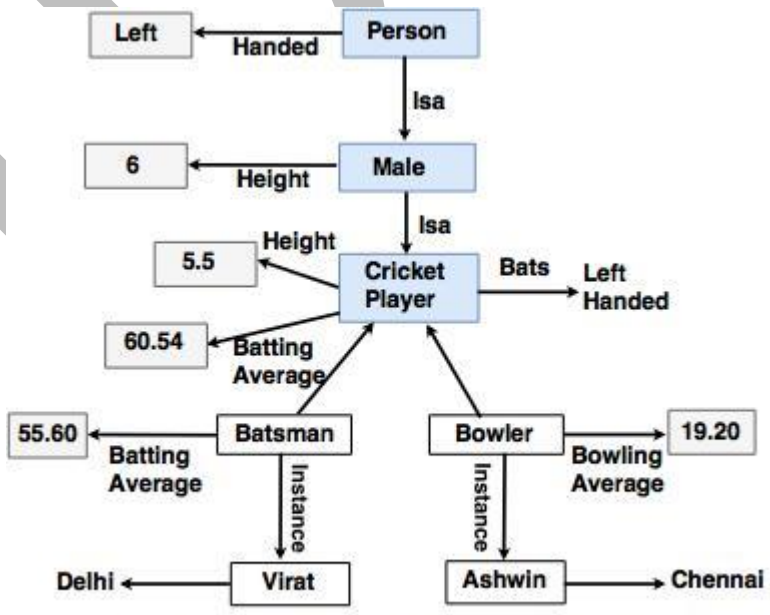


Fig: Inheratable Knowledge Representation

The above diagram to refer to the following issues.

1. Important attributes
There are two attributes shown in the diagram, **instance** and **isa.** Since these attributes support property of inheritance, they are of prime importance.

2. Relationships among attributes
Basically, the attributes used to describe objects are nothing but the entities. However, the attributes of an object do not depend on the encoded specific knowledge.

3. Choosing the granularity of representation
While deciding the granularity of representation, it is necessary to know the following:

i. What are the primitives and at what level should the knowledge be represented?

ii. What should be the number (small or large) of low-level primitives or high-level facts?
High-level facts may be insufficient to draw the conclusion while Low-level primitives may require a lot of storage.

4. Representing sets of objects.
There are some properties of objects which satisfy the condition of a set together but not as individual;

Example: Consider the assertion made in the sentences:
"There are more sheep than people in Australia", and "English speakers can be found all over the world." These facts can be described by including an assertion to the sets representing people, sheep, and English.

5. Finding the right structure as needed
To describe a particular situation, it is always important to find the access of right structure. This can be done by selecting an initial structure and then revising the choice.

2.5 Internal Representation

Internal symbolic **representation**: requires a common symbol language, in which an agent can express and manipulate propositions about the world. It is also known as FOL(first order logic)

- In order to act intelligently, a computer must have the knowledge about the domain of interest.
- Knowledge is the body of facts and principles gathered or the act, fact, or state of knowing.
- This knowledge needs to be presented in a form, which is understood by the machine.
- This unique format is called internal representation.
- Thus plain English sentences could be translated into an internal representation and they could be used to answer based on the given sentences.

2.5.1 Properties Of Internal Representation

An internal representation has following properties that has to be followed while implementing:

- Internal representation must remove all referential ambiguity.
  Referential ambiguity is the ambiguity about what the sentence refers to.

E.g. ‘Raj said that Ram was not well. He must be lying.’
Who does ‘he’ refer to?

- Internal representation should avoid word-sense ambiguity.
  Word-sense ambiguities arise because of multiple meaning of words.
  E.g. ‘Raj caught a pen. Raj caught a train. Raj caught fever.’
- Internal representation must explicitly mention functional structure.
  Functional structure is the word order used in the language to express an idea.
  E.g. ‘Ram killed Ravan. Ravan was killed by Ram.’
  Thus internal representation may not use the order of the original sentence.
- Internal representation should be able handle complex sentence without losing meaning attached with it.

2.6 Logic Representation

Facts are the general statements that may be either True or False. Thus, logic can be used to represent such simple facts.

To build a Logic-based representation:

- User has to define a set of primitive symbols along with the required semantics.
- The symbols are assigned together to define legal sentences in the language for representing TRUE facts.
- New logical statements are formed from the existing ones. The statements which can be either TRUE or false but not both , are called propositions. A declarative sentence expresses a statement with a proposition as content;
  Example: The declarative "Cotton is white" expresses that Cotton is white. So, the sentence "Cotton is white" is a true statement.

2.6.1 First Order Logic

- The prepositional logic only deals with the facts that may be true or false.
- The first order logic assumes that the world contains objects, relations and functions.

Syntax for first order logic:

- In prepositional logic, every expression is a sentence that represents a fact.
- First order logic includes the sentences along with terms which can represent the objects.
- Constant symbols, variables and function symbols are used to build terms, while quantifiers and predicate symbols are used to build the sentences.

2.6.2 Predicate Calculus

- Predicate Calculus is an internal representation methodology which helps us in deducing more results from the given propositions (statements).
- Predicate calculus access’s individual components of a proposition and represent the proposition.

- For example, the sentence 'Raj came late on Sunday' can be represented in predicate calculus as: (came-late Raj Sunday)
- Here 'came-late' is a predicate that describes the relation between a person and a day.
- 'Raj came late on a rainy Sunday' can be represented as: (came-late Raj Sunday) & (inst Sunday rainy)
- Predicate permits us to break a statement down into component parts namely, objects, a characteristic of the object, or some assertion about the object.

2.6.3 Syntax of Predicate Calculus

- Predicate and Arguments
    - In predicate calculus, a proposition is divided into two parts:
    - Arguments (or objects)
- Predicate (or assertion)
    - The arguments are the individual or objects an assertion is made about. The predicate is the assertion made about them.
    - In an English language sentence, objects are nouns that serve as subject and object of the sentence and predicate would be the verb or part of the verb.
    - For example the proposition: 'Vinod likes apple' would be stated as: (likes Vinod apple)
    - Where 'likes' is the predicate and Vinod and apple are the arguments.
    - In some cases, the proposition may not have any predicates. For example: Anita is a woman i.e. (inst Anita woman).
- Constants
    - Constants are fixed value terms that belong to a given domain.
    - They are denoted by numbers and words. E.g. 123, abc.
- Variables
    - In predicate calculus, letters may be substituted for the arguments.
    - The symbols x or y could be used to designate some object or individual.
    - The example "Vinod likes apple" could be expressed in variable form if x = Vinod and y = apple. Then the proposition becomes: (likes x, y)
    - If variables are used, then the stated proposition must be true for any names substituted for the variables.
- Instantiation
    - Instantiation is the process of assigning the name of a specific individual or object to a variable.
    - That object or individual becomes an "instance" of that variable.
    - In the previous example, supplying Vinod for x and apple for y is a case of instantiation.
- Connectives
    - There are four connectives used in predicate calculus.
    - They are 'not', 'and', 'or' and 'if'.
    - If p and q are formulas then (and p, q), (or p, q), (not p) and (if p, q) are also formulas.
    - They can be expressed in truth tables.
    - (not p):

p	(not p)
T	F
F	T

- (and p, q):

p	q	(and p, q)
T	T	T
T	F	F
F	T	F
F	F	F

- (or p, q):

p	q	(or p, q)
T	T	T
T	F	T
F	T	T
F	F	F

- (if p, q):

p	q	(if p, q)
T	T	T
T	F	F
F	T	T
F	F	T

- Quantifiers
  - A quantifier is a symbol that permits us to state the range or scope of the variables in a predicate logic expression.
  - Two quantifiers are used in logic:
  - The universal quantifier – 'for all'. E.g. (forall (x) f) for a formula f.
  - The existential quantifier – 'exists'. E.g. (exists (x) f) for a formula f.
- Function applications
  - It consists of a function which takes zero or more arguments.
  - E.g. friend-of (x).

- "All Maharashtrians are Indian citizens" could be expressed as:
  - (forall (x) (if Maharastrian(x) Indian citizen(x)).
- "Every car has a wheel" could be expressed as:

- (forall (x) (if (Car x) (exists (y) wheel-of (x y))).

The Predicate Calculus Consists Of:

- A set of constant terms.
- A set of variables.
- A set of predicates, each with a specified number of arguments.
- A set of functions, each with a specified number of arguments.
- The connectives - 'if', 'and', 'or' and 'not'.
- The quantifiers - 'exists' and 'forall'.
- The terms used in predicate calculus are:
  - Constant terms.
  - Variables.
  - Functions applied to the correct number of terms.
- **The formulas used in predicate calculus are:**
  - A predicate applied to the correct number of terms.
  - If p and q are formulas then (if p, q), (and p, q), (or p, q) and (not p).
  - If x is a variable, and p is a formula, then (exists(x) p), and (forall (x) p).
- In predicate calculus, the initial facts from which we can derive more facts are called axioms.
- The facts we deduce from the axioms are called theorems.
- The set of axioms are not stable and in fact change over time as new information (axioms) comes.

2.7 Inference Rules

A **rule of inference**, **inference rule** or **transformation rule** is a logical form consisting of a function which takes premises, analyzes their syntax, and returns a conclusion (or conclusions). For example, the rule of inference called *modus ponens* takes two premises, one in the form "If p then q" and another in the form "p", and returns the conclusion "q".

From a given set of axioms, we can deduce more facts using inference rules. The important inference rules are:

- Modus ponens: From p and (if p, q) infer q.
- Chain rule: From (if p, q) and (if q, r) infer (if p, r).
- Substitution: if p is a valid axiom, then a statement derived using consistent substitution of propositions is also valid.
- Simplification: From (and p, q) infer p.
- Conjunction: From p and q infer (and p q).
- Transposition: From (if p, q) infer (if (not q) (not p)).
- Universal instantiation: if something is true of everything, then it is true for any particular thing.
- Abduction: From q and (if p, q) infer p. (Abduction can lead to wrong conclusions. Still, it is very important as it gives lot explanation. For example: medical diagnosis.)
- Induction: From (P, a), (P, b) … infer (forall (x) (P, x)). (Induction leads to learning.)

2.7.1 Exercise: Express the Following in Predicate Calculus:-

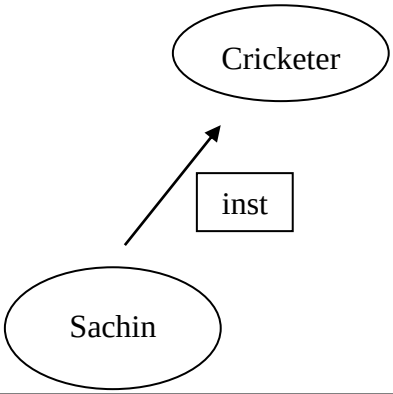
- Roses are red.
  Rose(x) -> Red(x,color)
- Violets are blue.
  Violets(x) -> Blue(x,color)
- Every chicken hatched from an egg.
  ∀(x), ∃y: chicken(x) ∧Egg(y) -> hatched(x,y)
- Some language is spoken by everyone in this class.
  ∀(x), ∃y: people(x) ∧ language(y) ∧ belongs(x,class) -> spoken(y,x)
- If you push anything hard enough, it will fall over.
  ∀(x): Any-object(x) ∧ push-hard(x) -> fall-over(x)
- Everybody loves somebody sometime.
  ∀(x), ∃y: loves-sometime(x, y))).
- Anyone with two or more spouses is a bigamist.
  (∀(x): ((inst x have-two-or-more-wife) (inst x bigamist)))

2.7.2 Alternative Notations

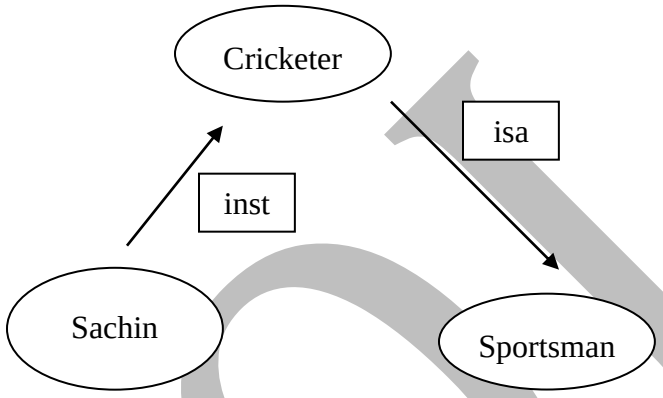
1. Semantic Net

A **semantic net** (or semantic network) is a knowledge representation technique used for propositional information. So it is also called a propositional net. Semantic nets convey meaning. They are two dimensional representations of knowledge. Mathematically a *semantic net* can be defined as a labelled directed graph. Semantic nets consist of nodes, links (edges) and link labels. In the semantic network diagram, nodes appear as circles or ellipses or rectangles to represent objects such as physical objects, concepts or situations. Links appear as arrows to express the relationships between objects, and link labels specify particular relations. Relationships provide the basic structure for organizing knowledge. The objects and relations involved need not be so concrete. As nodes are associated with other nodes semantic nets are also referred to as associative nets. They are basically graphical depictions of knowledge that show hierarchical relationships between objects.

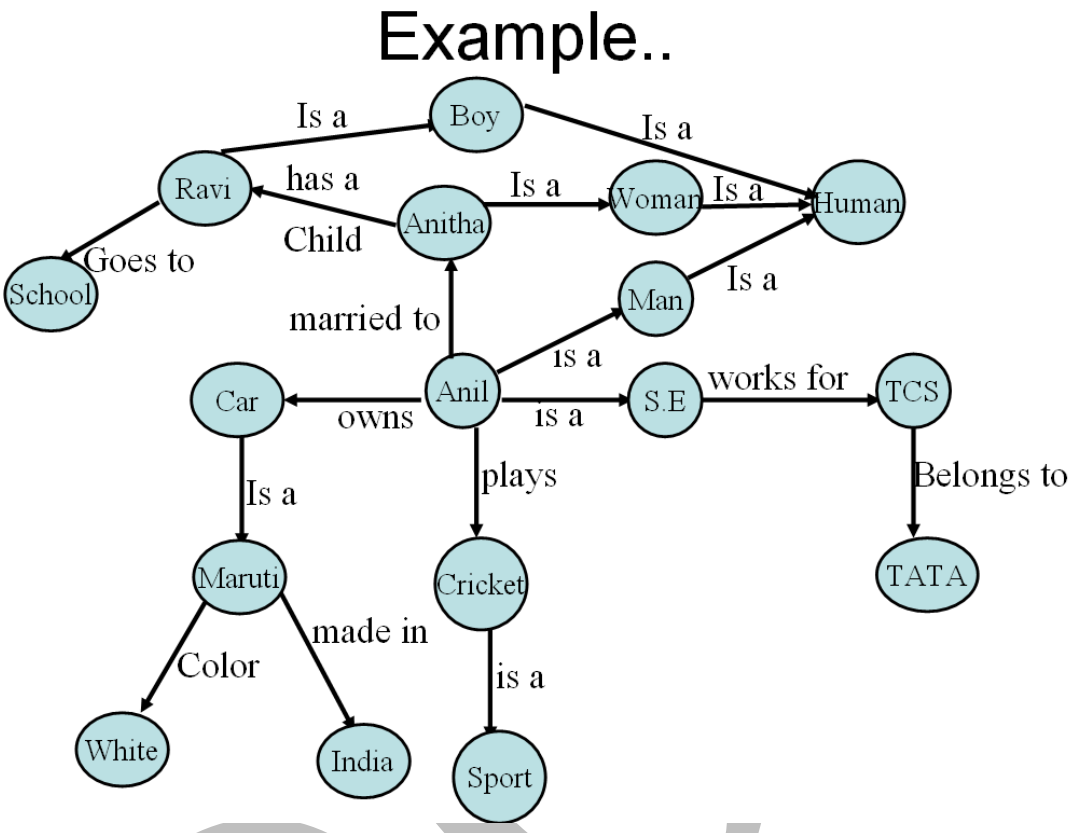
- For example 'Sachin is a cricketer'
- i.e. (inst Sachin cricketer), can be represented in associative network as



- A semantic network is made up of a number of ovals or circles called nodes.
- Nodes represent objects and descriptive information about those objects.
- Objects can be any physical item, concept, event or an action.
- The nodes are interconnected by links called arcs.
- These arcs show the relationships between the various objects and descriptive factors.
- The arrows on the lines point from an object to its value along the corresponding arc.



- E.g. (isa cricketer sportsman).
- The instance relation corresponds to the relation element-of.
- Sachin is an element of the set of cricketers. Thus he is an element of all the supersets of all cricketers.
- The 'isa' relation corresponds to the relation 'subset of'.
- Cricketers are a subset of sportsmen and hence cricketers inherit al the properties of sportsmen.



- The predicate calculus lacks a backward pointer resulting a long search for retrieving information.
- Thus the predicate calculus along with an indexing (pointing) scheme is a much better internal representation scheme than semantic networks as it has connectives and quantifiers.
- Slot Assertion Notation
- In a slot assertion notation various arguments, called slots, of predicate are expressed as separate assertions.
- Slot assertion notation is a special type of predicate calculus representation.
- For example (catch-object Sachin ball) can be expressed as
- (inst catch1 catch-object) … // catch1 is a one type of catching.
- (catcher catch1 Sachin) … // Sachin did the catching.
- (caught catch1 ball) … // he caught the ball.

Advantages of Semantic Nets

Semantic nets have the ability to represent default values for categories. In the above figure Jack has one leg while he is a person and all persons have two legs. So persons have two legs has only default status which can be overridden by a specific value.

- They convey some meaning in a transparent manner.
- The nets are simple and easy to understand.
- They are easy to translate into PROLOG.

Disadvantage of Semantic Nets

- One of the drawbacks of semantic network is that the links between the objects represent only binary relations. For example, the sentence Run(ChennaiExpress, Chennai,Bangalore,Today) cannot be asserted directly.
- There is no standard definition of link names.

2. Frame Notation

Natural language understanding requires inference i.e., assumptions about what is typically true of the objects or situations under consideration. Such information can be coded in structures known as frames.

Need of frames

Frame is a type of schema used in many AI applications including vision and natural language processing. Frames provide a convenient structure for representing objects that are typical to a stereotypical situations. The situations to represent may be visual scenes, structure of complex physical objects, etc. Frames are also useful for representing commonsense knowledge. As frames allow nodes to have structures they can be regarded as three-dimensional representations of knowledge.

A frame is similar to a record structure and corresponding to the fields and values are slots and slot fillers. Basically it is a group of slots and fillers that defines a stereotypical object. A single frame is not much useful. Frame systems usually have collection of frames connected to each other. Value of an attribute of one frame may be another frame.

Thus we have,

- (catch-object catch1 (catcher Sachin) (caught ball))

Here we have constructed a single structure called a frame that includes all the information.

Exercise: Convert The Following to First-Order Predicate Logic Using the Predicates Indicated:-

- swimming_pool(X)

  steamy(X)
  large(X)
  unpleasant(X)
  noisy(X)
  place(X)
- All large swimming pools are noisy and steamy places.
- All noisy and steamy places are unpleasant.

- All noisy and steamy places except swimming pools are unpleasant.
- The swimming pool is small and quiet.

Answers:-

- All large swimming pools are noisy and steamy places.
  (forall (x) (if (and large(X) swimming_pool(X)) (and noisy(X) (and (steamy(X) place(X))))).

- All noisy and steamy places are unpleasant.
  (forall (x) (if (and noisy(X) (and (steamy(X) place(X))) unpleasant(X))).

- All noisy and steamy places except swimming pools are unpleasant.
  (forall (x) (if ((not swimming_pool(x)) and noisy(X) (and (steamy(X) place(X))) unpleasant(X)))).

- The swimming pool is small and quiet.
  (and swimming_pool(x) and (not large(X)) (not noisy(X)))

Exercise:

1. What is knowledge? Explain its techqniue.
2. Discuss in brief about the historical evolution of Artificial Intelligence.
3. Explain the properties of Internal Representation.
4. What is Semantic Net? Explain with suitable example.
5. What is Predicate? Explain the components of predicates.
6. What are quantifiers? Give Suitable example for each.
7. Explain the concept of frames with suitable example.

Chapter 3

Introduction to LISP

1. Introduction

John McCarthy invented LISP in 1958, shortly after the development of FORTRAN. It was first implemented by Steve Russell on an IBM 704 computer. It is particularly suitable for Artificial Intelligence programs, as it processes symbolic information effectively. Common Lisp originated, during the 1980s and 1990s, in an attempt to unify the work of several implementation groups that were successors to Maclisp, like ZetaLisp and NIL (New Implementation of Lisp) etc. It serves as a common language, which can be easily extended for specific implementation. Programs written in Common LISP do not depend on machine-specific characteristics, such as word length etc.

Features of Common LISP

- It is machine-independent
- It uses iterative design methodology, and easy extensibility.
- It allows updating the programs dynamically.
- It provides high level debugging.
- It provides advanced object-oriented programming.
- It provides a convenient macro system.
- It provides wide-ranging data types like, objects, structures, lists, vectors, adjustable arrays, hash-tables, and symbols.
- It is expression-based.
- It provides an object-oriented condition system.
- It provides a complete I/O library.
- It provides extensive control structures.

2. Text Editor

This will be used to type your program. Examples of few editors include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.

Name and version of text editor can vary on different operating systems. For example, Notepad will be used on Windows, and vim or vi can be used on windows as well as Linux or UNIX.

The files you create with your editor are called source files and contain program source code. The source files for Lisp programs are typically named with the extension "**.lisp**".

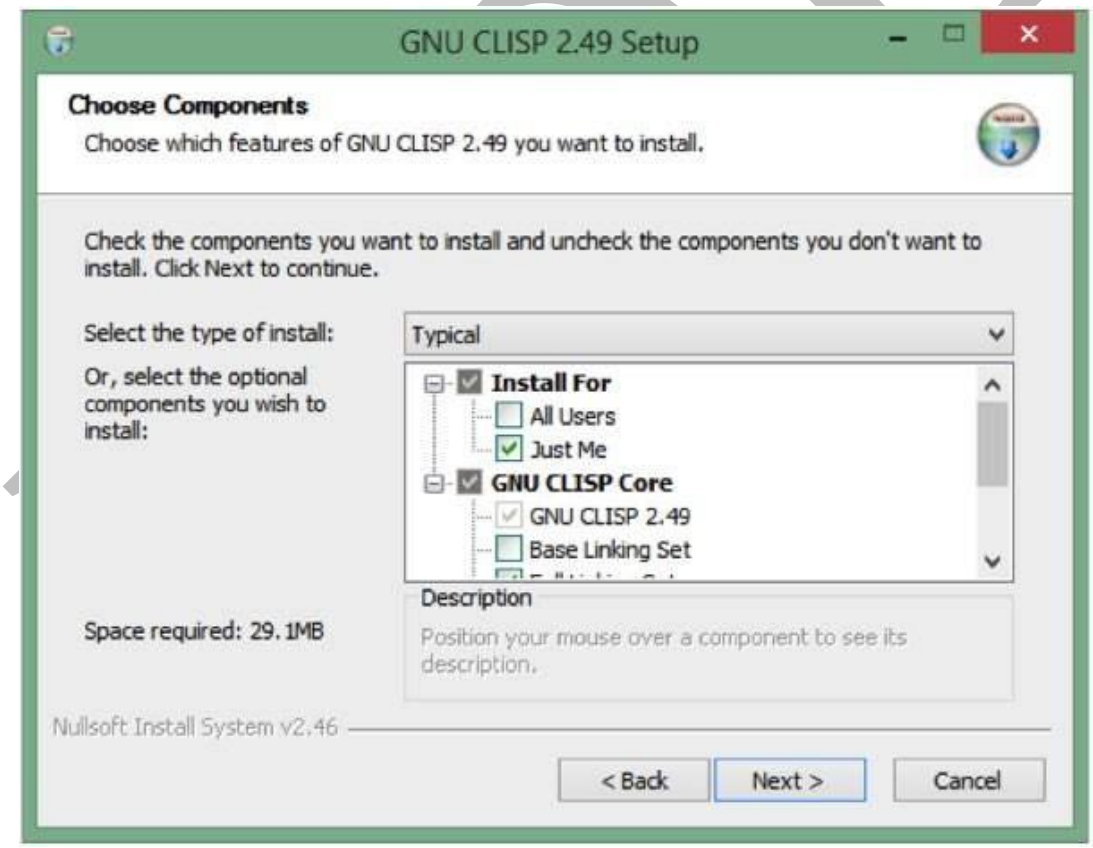
Before starting your programming, make sure you have one text editor in place and you have enough experience to write a computer program, save it in a file, finally execute it.

3. The Lisp Executer

The source code written in source file is the human readable source for your program. It needs to be "executed", to turn into machine language so that your CPU can actually execute the program as per instructions given.

This Lisp programming language will be used to execute your source code into final executable program. I assume you have basic knowledge about a programming language.

CLISP is the GNU Common LISP multi-architechtural compiler used for setting up LISP in Windows. The windows version emulates a unix environment using MingW under windows. The installer takes care of this and automatically adds clisp to the windows PATH variable.



LISP expressions are called symbolic expressions or s-expressions. The s-expressions are composed of three valid objects, atoms, lists and strings. Any s-expression is a valid program.

LISP programs run either on an **interpreter** or as **compiled code.** The interpreter checks the source code in a repeated loop, which is also called the read-evaluate-print loop (REPL). It reads the program code, evaluates it, and prints the values returned by the program.

4. A Simple Program

Let us write an s-expression to find the sum of three numbers 7, 9 and 11. To do this, we can type at the interpreter prompt.

```
(+7911)
```

LISP returns the result −

```
27
```

If you would like to run the same program as a compiled code, then create a LISP source code file named myprog.lisp and type the following code in it.

```
(write(+7911))
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is −

```
27
```

5. LISP Uses Prefix Notation

You might have noted that LISP uses **prefix notation.**

In the above program the + symbol works as the function name for the process of summation of the numbers.

In prefix notation, operators are written before their operands. For example, the expression,

```
a*( b + c )/ d
```

will be written as −

```
(/(* a (+ b c)) d)
```

Let us take another example, let us write code for converting Fahrenheit temp of 60° F to the centigrade scale −

The mathematical expression for this conversion will be −

```
(60*9/5)+32
```

Create a source code file named main.lisp and type the following code in it.

```
(write(+(*(/95)60)32))
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is−

```
140
```

6. Evaluation of LISP Programs

Evaluation of LISP programs has two parts −

- Translation of program text into Lisp objects by a reader program
- Implementation of the semantics of the language in terms of these objects by an evaluator program

The evaluation process takes the following steps −

- The reader translates the strings of characters to LISP objects or **s-expressions.**
- The evaluator defines syntax of Lisp **forms** that are built from s-expressions. This second level of evaluation defines a syntax that determines which **s-expressions** are LISP forms.
- The evaluator works as a function that takes a valid LISP form as an argument and returns a value. This is the reason why we put the LISP expression in parenthesis, because we are sending the entire expression/form to the evaluator as arguments.

The 'Hello World' Program

Learning a new programming language doesn't really take off until you learn how to greet the entire world in that language, right!

So, please create new source code file named main.lisp and type the following code in it.

```
(write-line"Hello World")

(write-line"I am at 'Tutorials Point'! Learning LISP")
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is −

```
Hello World

I am at 'Tutorials Point'! Learning LISP
```

7. LISP Expressions

When you start up the Common LISP environment, you should see a prompt that looks like the following:

```
USER(#):
```

OR

>

This means that LISP is waiting for you to enter a LISP expression. The Common LISP environment follows the algorithm below when interacting with users:

```
if(inputFromPrompt)
read an expression from the console;
evaluate the expression;
print the result of evaluation to console;
goto if.
```

Common LISP reads in an expression, evaluates it, and then prints out the result. For example, if you want to compute the value of *(2 *(4 + 6))*, you type in:

```
> (* 2 (+ 4 6))
```

It replies:

```
20
```

…before prompting you to enter the next expression.

This time we try (9000 + 900 + 90 + 9) − (5000 + 500 + 50 + 5):

```
  > (- (+ 9000 900 90 9) (+ 5000 500 50 5)) ⇒ 4444
```

Several things are worth noting:

- LISP expressions are composed of *forms*. The most common LISP form is *function application*. LISP represents a function call *f(x)* as `(f x)`. For example, *cos(0)* is written as `(cos 0)`.
- LISP expressions are case-insensitive. It makes no difference whether we type `(cos 0)` or `(COS 0)`.
- Similarly, "`+`" is the name of the addition function that returns the sum of its arguments.
- Some functions, like "`+`" and "`*`", could take an arbitrary number of arguments. In our example, "`*`" took three arguments. It could as well take 2 arguments, as in "`(* 2 3)`", or 4 arguments, as in "`(* 2 3 4 5)`".
- In general, a function application form looks like `(`*function argument*$_1$ *argument*$_2$ ... *argument*$_n$`)`. As in many programming languages (e.g. C/C++), LISP evaluates function calls in *applicative order (PEMDAS or BoDMAS)*, which means that all the argument forms are evaluated before the function is invoked. That is to say, the argument forms `(cos 0)` and `(+ 4 6)` are respectively evaluated to the values *1* and *10* before they are passed as arguments to the `*` function. Some other forms, like the *conditionals*, are not evaluated in applicative order.
- Numeric values like `4` and `6` are called *self-evaluating forms*: they evaluate to themselves. To evaluate `(+ 4 6)` in applicative order, the forms `4` and `6` are respectively evaluated to the values *4* and *6* before they are passed as arguments to the `+` function.

Complex arithmetic expressions can be constructed from built-in functions like the following:

Numeric Functions	Meaning
(+ x_1 x_2 ... x_n)	The sum of x_1, x_2, ..., x_n
(* x_1 x_2 ... x_n)	The product of x_1, x_2, ..., x_n
(- x y)	Subtract y from x
(/ x y)	Divide x by y
(rem x y)	The remainder of dividing x by y
(abs x)	The absolute value of x
(max x_1 x_2 ... x_n)	The maximum of x_1, x_2, ..., x_n
(min x_1 x_2 ... x_n)	The minimum of x_1, x_2, ..., x_n

Common LISP has a rich set of pre-defined numerical functions. For a complete coverage, consult the book, *Common LISP, The Language (2nd Edition)* (CLTL2) by Guy Steele which can be found online at:
https://www.cs.cmu.edu/Groups/AI/util/html/cltl/clm/node1.html

8. Basic Building Blocks in LISP

LISP programs are made up of three basic building blocks −

- atom
- list
- string

An **atom** is a number or string of contiguous characters. It includes numbers and special characters.

Following are examples of some valid atoms −

```
hello-from-tutorials-point
name
123008907
*hello*
Block#221
abc123
```

A **list** is a sequence of atoms and/or other lists enclosed in parentheses.

Following are examples of some valid lists −

```
(i am a list)
(a( a b c) d e fgh)
```

```
(father tom (susan bill joe))
(sun mon tue wed thurfri sat)
()
```

A **string** is a group of characters enclosed in double quotation marks.

Following are examples of some valid strings −

```
" I am a string"
"aba c d efg #$%^&!"
"Please enter the following details :"
"Hello from 'Tutorials Point'! "
```

9. Adding Comments

The semicolon symbol (;) is used for indicating a comment line.

For Example,

```
(write-line "Hello World"); greet the world

; tell them your whereabouts

(write-line"I am at 'Tutorials Point'! Learning LISP")
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is −

```
Hello World

I am at 'Tutorials Point'! Learning LISP
```

10.Some Notable Points before Moving to Next

Following are some of the important points to note −

- The basic numeric operations in LISP are +, -, *, and /
- LISP represents a function call f(x) as (f x), for example cos(45) is written as cos 45
- LISP expressions are case-insensitive, cos 45 or COS 45 are same.
- LISP tries to evaluate everything, including the arguments of a function. Only three types of elements are constants and always return their own value
    - Numbers
    - The letter **t,** that stands for logical true.
    - The value **nil,** that stands for logical false, as well as an empty list.

11.Data types

LISP data types can be categorized as.

- **Scalar types** − for example, number types, characters, symbols etc.
- **Data structures** − for example, lists, vectors, bit-vectors, and strings.

Any variable can take any LISP object as its value, unless you have declared it explicitly.

Although, it is not necessary to specify a data type for a LISP variable, however, it helps in certain loop expansions, in method declarations and some other situations that we will discuss in later chapters.

The data types are arranged into a hierarchy. A data type is a set of LISP objects and many objects may belong to one such set.

The **typep** predicate is used for finding whether an object belongs to a specific type.

The **type-of** function returns the data type of a given object.

Type Specifiers in LISP

Type specifiers are system-defined symbols for data types.

array	fixnum	package	simple-string
atom	float	pathname	simple-vector
bignum	function	random-state	single-float
bit	hash-table	ratio	standard-char
bit-vector	integer	rational	stream
character	keyword	readtable	string
[common]	list	sequence	[string-char]
compiled-function	long-float	short-float	symbol
complex	nill	signed-byte	t
cons	null	simple-array	unsigned-byte
double-float	number	simple-bit-vector	vector

Apart from these system-defined types, you can create your own data types. When a structure type is defined using **defstruct** function, the name of the structure type becomes a valid type symbol.

Example 1

Create new source code file named main.lisp and type the following code in it.

```
(setq x 10)
(setq y 34.567)
(setqchnil)
(setq n 123.78)
(setqbg11.0e+4)
(setq r 124/2)

(print x)
(print y)
(print n)
(printch)
(printbg)
(print r)
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is −

```
10
34.567
123.78
NIL
110000.0
62
```

Example 2

Next let's check the types of the variables used in the previous example. Create new source code file named main. lisp and type the following code in it.

```
(defvar x 10)
(defvar y 34.567)
(defvarchnil)
(defvar n 123.78)
(defvarbg11.0e+4)
(defvar r 124/2)

(print(type-of x))
(print(type-of y))
(print(type-of n))
(print(type-of ch))
(print(type-of bg))
(print(type-of r))
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is −

```
(INTEGER 0 281474976710655)
SINGLE-FLOAT
SINGLE-FLOAT
```

```
NULL
SINGLE-FLOAT
(INTEGER 0 281474976710655)
```

12.Defining a Macro

In LISP, a named macro is defined using another macro named **defmacro.** Syntax for defining a macro is −

```
(defmacro macro-name (parameter-list))
"Optional documentation string."
body-form
```

The macro definition consists of the name of the macro, a parameter list, an optional documentation string, and a body of Lisp expressions that defines the job to be performed by the macro.

Example

Let us write a simple macro named setTo10, which will take a number and set its value to 10.

Create new source code file named main.lisp and type the following code in it.

```
defmacro setTo10(num)
(setqnum10)(printnum))
(setq x 25)
(print x)
(setTo10 x)
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is −

```
25
10
```

13. Variables

In LISP, each variable is represented by a **symbol**. The variable's name is the name of the symbol and it is stored in the storage cell of the symbol.

Global Variables

Global variables have permanent values throughout the LISP system and remain in effect until a new value is specified.

Global variables are generally declared using the **defvar** construct.

For example

```
(defvar x 234)
 (write x)
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is 234

Since there is no type declaration for variables in LISP, you directly specify a value for a symbol with the **setq** construct.

For Example

```
->(setq x 10)
```

The above expression assigns the value 10 to the variable x. You can refer to the variable using the symbol itself as an expression.

The **symbol-value** function allows you to extract the value stored at the symbol storage place.

For Example

Create new source code file named main.lisp and type the following code in it.

```
(setq x 10)
(setq y 20)
(format t "x = ~2d y = ~2d ~%" x y)
(setq x 100)
(setq y 200)
(format t "x = ~2d y = ~2d" x y)
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is.

```
x = 10 y = 20
x = 100 y = 200
```

Local Variables

Local variables are defined within a given procedure. The parameters named as arguments within a function definition are also local variables. Local variables are accessible only within the respective function.

Like the global variables, local variables can also be created using the **setq** construct.

There are two other constructs - **let** and **prog** for creating local variables.

The let construct has the following syntax.

```
(let((var1  val1)(var2  val2)..(varnvaln))<s-expressions>)
```

Where var1, var2, ..varn are variable names and val1, val2, .. valn are the initial values assigned to the respective variables.

When **let** is executed, each variable is assigned the respective value and lastly the *s-expression* is evaluated. The value of the last expression evaluated is returned.

If you don't include an initial value for a variable, it is assigned to **nil.**

Example

Create new source code file named main.lisp and type the following code in it.

```
(let((x 'a) (y 'b)(z 'c))
(format t "x = ~a y = ~a z = ~a" x y z))
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is.

```
x = A y = B z = C
```

The **prog** construct also has the list of local variables as its first argument, which is followed by the body of the **prog,** and any number of s-expressions.The **prog** function executes the list of s-expressions in sequence and returns nil unless it encounters a function call named **return.** Then the argument of the **return** function is evaluated and returned.

Example

Create new source code file named main.lisp and type the following code in it.

```
(prog((x '(a b c))(y '(123))(z '(p q 10)))
(format t "x = ~a y = ~a z = ~a" x y z))
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is.

```
x = (A B C) y = (1 2 3) z = (P Q 10)
```

14. Constants

In LISP, constants are variables that never change their values during program execution. Constants are declared using the defconstant construct.

Example

The following example shows declaring a global constant PI and later using this value inside a function named *area-circle* that calculates the area of a circle.

The **defun** construct is used for defining a function, we will look into it in the **Functions** chapter.

Create a new source code file named main.lisp and type the following code in it.

```
(defconstant PI 3.141592)
(defun area-circle(rad)
(terpri)
(format t "Radius: ~5f" rad)
(format t "~%Area: ~10f"(* PI rad rad)))
(area-circle10)
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is.

```
Radius:  10.0
Area:   314.1592
```

15. Control Structures: Recursions and Conditionals

Now that we are equipped with all the tools for developing LISP programs, let us venture into something more interesting. Consider the definition of factorials:

n! = 1 if *n = 1*
*n! = n * (n - 1)!* if *n > 1*

We can implement a function to compute factorials using recursion:

```
(defun factorial (N)
  "Compute the factorial of N."
  (if (= N 1)
      1
    (* N (factorial (- N 1)))))
```

The `if` form checks if `N` is one, and returns one if that is the case, or else returns *N * (N - 1)!*. Several points are worth noting:

- The condition expression `(= N 1)` is a relational expression. It returns boolean values `T` or `NIL`. In fact, LISP treats `NIL` as false, and everything else as true. Other relational operators include the following:

Relational Operators	Meaning
`(= x y)`	*x* is equal to *y*
`(/= x y)`	*x* is not equal to *y*
`(< x y)`	*x* is less than *y*
`(> x y)`	*x* is greater than *y*
`(<= x y)`	*x* is no greater than *y*

`(>= x y)`	*x* is no less than *y*

- The `if` form is not a *strict function* (strict functions evaluate their arguments in applicative order). Instead, the `if` form evaluates the condition `(= N 1)` before further evaluating the other two arguments. If the condition evaluates to true, then only the second argument is evaluated, and its value is returned as the value of the `if` form. Otherwise, the third argument is evaluated, and its value is returned. Forms that are not strict functions are called *special forms*.
- The function is recursive. The definition of `factorial` involves invocation of itself. Recursion is, for now, our only mechanism for producing looping behavior. Specifically, the kind of recursion we are looking at is called *linear recursion*, in which the function may make at most one recursive call from any level of invocation.

To better understand the last point, we can make use of the debugging facility `trace` (do not compile your code if you want to use `trace`):

```
> (trace factorial)
(FACTORIAL)
> (factorial 4)
 0: (FACTORIAL 4)
   1: (FACTORIAL 3)
     2: (FACTORIAL 2)
       3: (FACTORIAL 1)
       3: returned 1
     2: returned 2
   1: returned 6
 0: returned 24
24
```

Tracing `factorial` allows us to examine the recursive invocation of the function. As you can see, at most one recursive call is made from each level of invocation.

Multiple Recursions

Recall the definition of Fibonacci numbers:

Fib(n) = 1 for *n = 0* or *n = 1*
Fib(n) = Fib(n-1) + Fib(n-2) for *n > 1*

This definition can be directly translated to the following LISP code:

```
(defunfibonacci (N)
  "Compute the N'th Fibonacci number."
  (if (or (zerop N) (= N 1))
      1
    (+ (fibonacci (- N 1)) (fibonacci (- N 2)))))
```

Again, several observations can be made. First, the function call `(zerop N)` tests if `N` is zero. It is merely a shorthand for `(= N 0)`. As such, `zerop` returns either `T` or `NIL`. We call such a boolean function a *predicate*, as indicated by the suffix `p`. Some other built-in shorthands and predicates are the following:

Shorthand	Meaning
`(1+ x)`	`(+ x 1)`
`(1- x)`	`(- x 1)`
`(zerop x)`	`(= x 0)`
`(plusp x)`	`(> x 0)`
`(minusp x)`	`(< x 0)`
`(evenp x)`	`(= (rem x 2) 0)`
`(oddp x)`	`(/= (rem x 2) 0)`

Second, the `or` form is a logical operator. Like `if`, `or` is not a strict function. It evaluates its arguments from left to right, returning non-`NIL` immediately if it encounters an argument that evaluates to non-`NIL`. It evaluates to `NIL` if all tests fail. For example, in the expression `(or t (= 1 1))`, the second argument `(= 1 1)` will not be evaluated. Similar logical connectives are listed below:

Logical Operators	Meaning
`(or` x_1 x_2 `...` x_n`)`	Logical or
`(and` x_1 x_2 `...` x_n`)`	Logical and
`(not x)`	Logical negation

Third, the function definition contains two self-references. It first recursively evaluates `(fibonacci (- N 1))` to compute *Fib(N-1)*, then evaluates `(fibonacci (- N 2))` to obtain *Fib(N-2)*, and lastly return their sum. This kind of recursive definitiion is called *double recursion* (more generally, *multiple recursion*). Tracing the function yields the following:

```
> (fibonacci 3)
 0: (FIBONACCI 3)
   1: (FIBONACCI 2)
     2: (FIBONACCI 1)
     2: returned 1
     2: (FIBONACCI 0)
     2: returned 1
   1: returned 2
   1: (FIBONACCI 1)
   1: returned 1
 0: returned 3
3
```

Note that in each level, there could be up to two recursive invocations.

Some beginners might find nested function calls like the following very difficult to understand:

```
(+ (fibonacci (- N 1)) (fibonacci (- N 2)))))
```

To make such expressions easier to write and comprehend, one could define local name bindings to represent intermediate results:

```
(let
    ((F1 (fibonacci (- N 1)))
     (F2 (fibonacci (- N 2))))
  (+ F1 F2))
```

The `let` special form above defines two local variables, `F1` and `F2`, which binds to *Fib(N-1)* and *Fib(N-2)* respectively. Under these local bindings, `let` evaluates `(+ F1 F2)`. The `fibonacci`function can thus be rewritten as follows:

```
(defunfibonacci (N)
  "Compute the N'th Fibonacci number."
  (if (or (zerop N) (= N 1))
      1
    (let
        ((F1 (fibonacci (- N 1)))
         (F2 (fibonacci (- N 2))))
      (+ F1 F2))))
```

Notice that `let` creates all bindings in parallel. That is, both `(fibonacci (- N 1))` and `(fibonacci (- N 2))` are evaluated first, and then they are bound to `F1` and `F2`. This means that the following LISP code will not work:

```
(let
    ((x 1)
     (y (* x 2)))
  (+ x y))
```

LISP will attempt to evaluate the right hand sides first before the bindings are established. So, the expression `(* x 2)` is evaluated before the binding of `x` is available. To perform sequential binding, use the `let*` form instead:

```
(let*
    ((x 1)
     (y (* x 2)))
  (+ x y))
```

LISP will bind `1` to `x`, then evaluate `(* x 2)` before the value is bound to `y`.

Exercise:

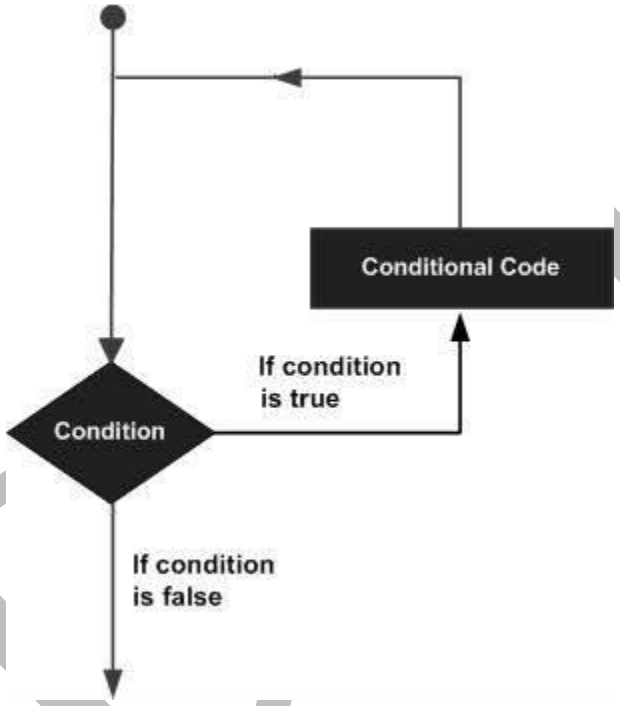
1. Explain the features of LISP.
2. Explain the structure of LISP program.
3. Explain how expressions are processed in LISP environment.
4. What is Macro? Explain its syntax.
5. Write a program to create macro for square of a number.
6. Explain different data types supported by lisp.
7. Explain how control structures are defined.

Chapter 4

LISP Functions

1. Loops

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages.



2. Defining Functions

Evaluating expressions is not very interesting. We would like to build expression abstractions that could be reused in the future. For example, we could type in the following:

```
> (defun double (x) (* x 2))
DOUBLE
```

In the above, we define a function named `double`, which returns two times the value of its input argument *x*. We can then test-drive the function as below:

```
> (double 3)
6
> (double 7)
14
```

3. Editing, Loading and Compiling LISP Programs

Most of the functions we would like to write are going to be a lot longer than the `double` function. When working with complex programs, it is usually desirable to edit the program with an editor, fully debug the code, and then compile it for faster performance.

```
(defun triple (X)
  (* 3 X))

(defun negate (X)
  (- X))
```

Save the above in the file `function.lisp`. Now load the definition into the LISP environment by typing:

```
> (load "function.lisp")
; Loading ./function.lisp
T
```

Let us try to see if they are working properly.

```
> (triple 2)
6
> (negate 3)
-3
```

When functions are fully debugged, we can also compile them into binaries:

```
> (compile-file "function.lisp")
```

4. Lists

Numeric values are not the only type of data LISP supports. LISP is designed for symbolic computing. The fundamental LISP data structure for supporting symbolic manipulation are lists. In fact, LISP stands for "LISt Processing."

Lists are containers that supports sequential traversal. List is also a *recursive data structure*: its definition is recursive. As such, most of its traversal algorithms are recursive functions. In order to better understand a recursive abstract data type and prepare oneself to develop recursive operations on the data type, one should present the data type in terms of its *constructors*, *selectors* and *recognizers*.

Constructors are forms that create new instances of a data type (possibly out of some simpler components). A list is obtained by evaluating one of the following constructors:

1. `nil`: Evaluating `nil` creates an *empty* list;
2. `(cons x L)`: Given a LISP object *x* and a list *L*, evaluating `(cons x L)` creates a list containing *x* followed by the elements in *L*.

Notice that the above definition is inherently recursive. For example, to construct a list containing 1 followed by 2, we could type in the expression:

```
> (cons 1 (cons 2 nil))
(1 2)
```
LISP replies by printing `(1 2)`, which is a more readable representation of a list containing 1 followed by 2. To understand why the above works, notice that `nil` is a list (an empty one), and thus `(cons 2 nil)` is also a list (a list containing 1 followed by nothing). Applying the second constructor again, we see that `(cons 1 (cons 2 nil))` is also a list (a list containing 1 followed by 2 followed by nothing).

Typing `cons` expressions could be tedious. If we already know all the elements in a list, we could enter our list as *list literals*. For example, to enter a list containing all prime numbers less than 20, we could type in the following expression:

```
> (quote (2 3 5 7 11 13 17 19))
(2 3 5 7 11 13 17 19)
```

Notice that we have quoted the list using the `quote` special form. This is necessary because, without the quote, LISP would interpret the expression `(2 3 5 7 11 13 17 19)` as a function call to a function with name "2" and arguments 3, 5, ..., 19 The `quote` is just a syntactic device that instructs LISP not to evaluate the a form in applicative order, but rather treat it as a literal. Since quoting is used frequently in

LISP programs, there is a shorthand for `quote`:

```
> '(2 3 5 7 11 13 17 19)
(2 3 5 7 11 13 17 19)
```

The quote symbol `'` is nothing but a syntactic shorthand for `(quote ...)`.

The second ingredient of an abstract data type are its selectors. Given a composite object constructed out of several components, a selector form returns one of its components. Specifically, suppose a list L_1 is constructed by evaluating `(cons x` L_2`)`, where x is a LISP object and L_2 is a list. Then, the selector forms `(first` L_1`)` and `(rest` L_1`)` evaluate to x and L_2 respectively, as the following examples illustrate:

```
> (first '(2 4 8))
2
> (rest '(2 4 8))
(4 8)
> (first (rest '(2 4 8)))
4
> (rest (rest '(2 4 8)))
(8)
> (rest (rest (rest '(8))))
NIL
```

Finally, we look at recognizers, expressions that test how an object is constructed. Corresponding to each constructor of a data type is a recognizer. In the case of list,

they are `null` for `nil` and `consp` for `cons`. Given a list *L*, `(null L)` returns `t` iff *L* is `nil`, and `(consp L)` returns `t` iff *L* is constructed from `cons`.

```
> (null nil)
T
> (null '(1 2 3))
NIL
> (consp nil)
NIL
> (consp '(1 2 3))
T
```

Notice that, since lists have only two constructors, the recognizers are complementary. Therefore, we usually need only one of them. In our following discussion, we use only `null`.

5. Structural Recursion with Lists

As we have promised, understanding how the constructors, selectors and recognizers of lists work helps us to develop recursive functions that traverse a list. Let us begin with an example. The LISP built-in function `list-length` counts the number of elements in a list. For example,

```
> (list-length '(2 3 5 7 11 13 17 19))
8
```

Let us try to see how such a function can be implemented recursively. A given list *L* is created by either one of the two constructors, namely `nil` or a `cons`:

- *Case 1*: *L* is `nil`.
  The length of an empty list is zero.
- *Case 2*: *L* is constructed by `cons`.
  Then *L* is composed of two parts, namely, `(first L)` and `(rest L)`. In such case, the length of *L* can be obtained inductively by adding 1 to the length of `(rest L)`.

Formally, we could implement our own version of `list-length` as follows:

```
(defun recursive-list-length (L)
  "A recursive implementation of list-length."
  (if (null L)
      0
    (1+ (recursive-list-length (rest L)))))
```

Here, we use the recognizer `null` to differentiate how *L* is constructed. In case *L* is `nil`, we return 0 as its length. Otherwise, *L* is a `cons`, and we return 1 plus the length of `(rest L)`. Recall that `(1+ n)` is simply a shorthand for `(+ n 1)`.

Again, it is instructive to use the trace facilities to examine the unfolding of recursive invocations:

```
> (trace recursive-list-length)
(RECURSIVE-LIST-LENGTH)
> (recursive-list-length '(2 3 5 7 11 13 17 19))
 0: (RECURSIVE-LIST-LENGTH (2 3 5 7 11 13 17 19))
   1: (RECURSIVE-LIST-LENGTH (3 5 7 11 13 17 19))
     2: (RECURSIVE-LIST-LENGTH (5 7 11 13 17 19))
       3: (RECURSIVE-LIST-LENGTH (7 11 13 17 19))
         4: (RECURSIVE-LIST-LENGTH (11 13 17 19))
           5: (RECURSIVE-LIST-LENGTH (13 17 19))
             6: (RECURSIVE-LIST-LENGTH (17 19))
               7: (RECURSIVE-LIST-LENGTH (19))
                 8: (RECURSIVE-LIST-LENGTH NIL)
                 8: returned 0
               7: returned 1
             6: returned 2
           5: returned 3
         4: returned 4
       3: returned 5
     2: returned 6
   1: returned 7
 0: returned 8
8
```

The kind of recursion we see here is called *structural recursion*. Its standard pattern is as follows. To process an instance *X* of a recursive data type:

1. Use the recognizers to determine how *X* is created (i.e. which constructor creates it). In our example, we use `null` to decide if a list is created by `nil` or `cons`.
2. For instances that are atomic (i.e. those created by constructors with no components), return a trivial value. For example, in the case when a list is `nil`, we return zero as its length.
3. If the instance is composite, then use the selectors to extract its components. In our example, we use `first` and `rest` to extract the two components of a nonempty list.
4. Following that, we apply recursion on one or more components of *X*. For instance, we recusively invoked `recursive-list-length` on `(rest L)`.
5. Finally, we use either the constructors or some other functions to combine the result of the recursive calls, yielding the value of the function. In the case of `recursive-list-length`, we return one plus the result of the recursive call.

Sometimes, long traces like the one for list-length may be difficult to read on a terminal screen. Common LISP allows you to capture screen I/O into a file so that you can, for example, produce a hard copy for more comfortable reading. To capture the

trace of executing `(recursive-list-length '(2 3 5 7 11 13 17 19))`, we use the `dribble` command:

```
> (dribble "output.txt")
dribbling to file "output.txt"

NIL
> (recursive-list-length '(2 3 5 7 11 13 17 19))
 0: (RECURSIVE-LIST-LENGTH (2 3 5 7 11 13 17 19))
   1: (RECURSIVE-LIST-LENGTH (3 5 7 11 13 17 19))
     2: (RECURSIVE-LIST-LENGTH (5 7 11 13 17 19))
       3: (RECURSIVE-LIST-LENGTH (7 11 13 17 19))
         4: (RECURSIVE-LIST-LENGTH (11 13 17 19))
           5: (RECURSIVE-LIST-LENGTH (13 17 19))
             6: (RECURSIVE-LIST-LENGTH (17 19))
               7: (RECURSIVE-LIST-LENGTH (19))
                 8: (RECURSIVE-LIST-LENGTH NIL)
                 8: returned 0
               7: returned 1
             6: returned 2
           5: returned 3
         4: returned 4
       3: returned 5
     2: returned 6
   1: returned 7
 0: returned 8
8
> (dribble)
```

The form `(dribble "output.txt")` instructs Common LISP to begin capturing all terminal I/O into a file called `output.txt`. The trailing `(dribble)` form instructs Common LISP to stop I/O capturing, and closes the file `output.txt`. If we examine `output.txt`, we will see the following:

```
dribbling to file "output.txt"

NIL
> (recursive-list-length '(2 3 5 7 11 13 17 19))
 0: (RECURSIVE-LIST-LENGTH (2 3 5 7 11 13 17 19))
   1: (RECURSIVE-LIST-LENGTH (3 5 7 11 13 17 19))
     2: (RECURSIVE-LIST-LENGTH (5 7 11 13 17 19))
       3: (RECURSIVE-LIST-LENGTH (7 11 13 17 19))
         4: (RECURSIVE-LIST-LENGTH (11 13 17 19))
           5: (RECURSIVE-LIST-LENGTH (13 17 19))
             6: (RECURSIVE-LIST-LENGTH (17 19))
               7: (RECURSIVE-LIST-LENGTH (19))
                 8: (RECURSIVE-LIST-LENGTH NIL)
                 8: returned 0
               7: returned 1
             6: returned 2
           5: returned 3
         4: returned 4
       3: returned 5
     2: returned 6
   1: returned 7
 0: returned 8
```

```
8
> (dribble)
```

6. Symbols

The lists we have seen so far are lists of numbers. Another data type of LISP is *symbols*. A symbol is simply a sequence of characters:

```
> 'a           ; LISP is case-insensitive.
A
> 'A           ; 'a and 'A evaluate to the same symbol.
A
> 'apple2      ; Both alphanumeric characters ...
APPLE2
> 'an-apple    ; ... and symbolic characters are allowed.
AN-APPLE
>t            ; Our familiar t is also a symbol.
T
> 't           ; In addition, quoting is redundant for t.
T
>nil          ; Our familiar nil is also a symbol.
NIL
> 'nil         ; Again, it is self-evaluating.
NIL
```

With symbols, we can build more interesting lists:

```
> '(how are you today ?)   ; A list of symbols.
(HOW ARE YOU TODAY ?)
> '(1 + 2 * x)             ; A list of symbols and numbers.
(1 + 2 * X)
> '(pair (2 3))            ; A list containing 'pair and '(2 3).
(pair (2 3))
```

Notice that the list `(pair (2 3))` has length 2:

```
> (recursive-list-length '(pair (2 3)))
2
```

Notice also the result of applying accessors:

```
> (first '(pair (2 3)))
PAIR
> (rest '(pair (2 3)))
((2 3))
```

Lists containing other lists as members are difficult to understand for beginners. Make sure you understand the above example.

Example: `nth`

LISP defines a function `(nth N L)` that returns the *N*'th member of list *L* (assuming that the elements are numbered from zero onwards):

```
> (nth 0 '(a b c d))
```

```
A
> (nth 2 '(a b c d))
C
```

We could implement our own version of `nth` by linear recursion. Given *N* and *L*, either *L* is `nil` or it is constructed by `cons`.

- *Case 1*: *L* is `nil`.
  Accessing the *N*'th element is an undefined operation, and our implementation should arbitrarily return `nil` to indicate this.
- *Case 2*: *L* is constructed by a `cons`.
  Then *L* has two components: `(first L)` and `(rest L)`. There are two subcases: either *N = 0* or *N > 0*:
  - *Case 2.1*: *N = 0*.
    The zeroth element of *L* is simply `(first L)`.
  - *Case 2.2*: *N > 0*.
    The *N*'th member of *L* is exactly the *(N-1)*'th member of `(rest L)`.

The following code implements our algorithm:

```
(defun list-nth (N L)
  "Return the N'th member of a list L."
  (if (null L)
nil
    (if (zerop N)
        (first L)
      (list-nth (1- N) (rest L)))))
```

Recall that `(1- N)` is merely a shorthand for `(- N 1)`. Notice that both our implementation and its correctness argument closely follow the standard pattern of structural recursion. Tracing the execution of the function, we get:

```
> (list-nth 2 '(a b c d))
 0: (LIST-NTH 2 (A B C D))
   1: (LIST-NTH 1 (B C D))
     2: (LIST-NTH 0 (C D))
     2: returned C
   1: returned C
 0: returned C
C
```

Notice that we have a standard if-then-else-if structure in our implementation of `list-nth`. Such logic can alternatively be implemented using the `cond` special form.

```
(defun list-nth (n L)
  "Return the n'th member of a list L."
  (cond
   ((null L)   nil)
   ((zerop n)  (first L))
   (t          (list-nth (1- n) (rest L)))))
```

The `cond` form above is evaluated as follows. The condition `(null L)` is evaluated first. If the result is true, then `nil` is returned. Otherwise, the condition `(zerop n)` is

evaluated. If the condition holds, then the value of `(first L)` is returned. In case neither of the conditions holds, the value of `(list-nth (1- n) (rest L))` is returned.

Example: `member`

LISP defines a function `(member E L)` that returns non-NIL if *E* is a member of *L*.

```
> (member 'b '(perhaps today is a good day to die)) ; test fails
NIL
> (member 'a '(perhaps today is a good day to die)) ; returns non-NIL
'(a good day to die)
```

We implement our own recursive version as follows:

```
(defun list-member (E L)
  "Test if E is a member of L."
  (cond
   ((null L)          nil)
   ((eq E (first L))  t)
   (t                 (list-member E (rest L)))))
```

The correctness of the above implementation is easy to justify. The list *L* is either constructed by *nil* or by a call to *cons*:

- *Case 1*: *L* is `nil`.
  L is empty, and there is no way *E* is in *L*.
- *Case 2*: *L* is constructed by `cons`
  Then it has two components: `(first L)` and `(rest L)`. There are two cases, either `(first L)` is *E* itself, or it is not.
  - *Case 2.1*: *E* equals `(first L)`.
    This means that *E* is a member of *L*,
  - *Case 2.2*: *E* does not equal `(first L)`.
    Then *E* is a member of *L* iff *E* is a member of `(rest L)`.

Tracing the execution of `list-member`, we get the following:

```
> (list-member 'a '(perhaps today is a good day to die))
 0: (LIST-MEMBER A (PERHAPS TODAY IS A GOOD DAY TO DIE))
   1: (LIST-MEMBER A (TODAY IS A GOOD DAY TO DIE))
     2: (LIST-MEMBER A (IS A GOOD DAY TO DIE))
       3: (LIST-MEMBER A (A GOOD DAY TO DIE))
       3: returned T
     2: returned T
   1: returned T
 0: returned T
T
```

In the implementation of `list-member`, the function call `(eq x y)` tests if two symbols are the same. In fact, the semantics of this test determines what we mean by a member:

```
> (list-member '(a b) '((a a) (a b) (a c)))
 0: (LIST-MEMBER (A B) ((A A) (A B) (A C)))
   1: (LIST-MEMBER (A B) ((A B) (A C)))
     2: (LIST-MEMBER (A B) ((A C)))
       3: (LIST-MEMBER (A B) NIL)
       3: returned NIL
     2: returned NIL
   1: returned NIL
 0: returned NIL
NIL
```

In the example above, we would have expected a result of `t`. However, since `'(a b)` does not `eq` another copy of `'(a b)` (they are not the same symbol), `list-member` returns `nil`. If we want to account for list equivalence, we could have used the LISP built-in function `equal` instead of `eq`. Common LISP defines the following set of predicates for testing equality:

`(= x y)`	True if *x* and *y* evaluate to the same number.
`(eq x y)`	True if *x* and *y* evaluate to the same symbol.
`(eql x y)`	True if *x* and *y* are either `=` or `eq`.
`(equal x y)`	True if *x* and *y* are `eql` or if they evaluate to the same list.
`(equalp x y)`	To be discussed in Tutorial 4.

Example: `append`

LISP defines a function `append` that appends one list by another:

```
> (append '(a b c) '(c d e))
(A B C C D E)
```

We implement a recursive version of `append`. Suppose we are given two lists *L1* and *L2*. *L1* is either `nil` or constructed by `cons`.

- *Case 1*: *L1* is `nil`.
  Appending *L2* to *L1* simply results in *L2*.
- *Case 2*: *L1* is composed of two parts: `(first L1)` and `(rest L1)`. If we know the result of appending *L2* to `(rest L1)`, then we can take this result,
  insert `(first L1)` to the front, and we then have the list we want.

Formally, we define the following function:

```
(defun list-append (L1 L2)
  "Append L1 by L2."
  (if (null L1)
      L2
    (cons (first L1) (list-append (rest L1) L2))))
```

An execution trace is the following:

```
> (list-append '(a b c) '(c d e))
 0: (LIST-APPEND (A B C) (C D E))
```

```
   1: (LIST-APPEND (B C) (C D E))
     2: (LIST-APPEND (C) (C D E))
       3: (LIST-APPEND NIL (C D E))
       3: returned (C D E)
     2: returned (C C D E)
   1: returned (B C C D E)
 0: returned (A B C C D E)
(A B C C D E)
```

7. Using Lists as Sets

Formally, lists are ordered sequences. They differ with sets in two ways:

1. Sets are unordered, but lists are. `(a b c)` and `(c b a)` are two different lists.
2. An element either belong to a set or it does not. There is no notion of multiple occurrences. Yet, a list may contain multiple occurrences of the same element. `(a b b c)` and `(a b c)` are two different lists.

However, one may use lists to approximate sets, although the performance of such implementation is not the greatest.

We have already seen how we can use the built-in function `member` to test set membership. LISP also defines functions
like `(intersection` *`L1 L2`*`)`, `(union` *`L1 L2`*`)` and `(difference` *`L1 L2`*`)` for boolean operations on sets. In fact, these functions are not difficult to implement. Consider the following implementation of set intersection:

```
(defun list-intersection (L1 L2)
  "Return a list containing elements belonging to both L1 and L2."
  (cond
   ((null L1) nil)
   ((member (first L1) L2)
    (cons (first L1) (list-intersection (rest L1) L2)))
   (t (list-intersection (rest L1) L2))))
```

The correctness of the implementation is easy to see. *L1* is either an empty set (`nil`) or it is not:

- *Case 1*: *L1* is an empty set.
  Then its interection with *L2* is obviously empty.
- *Case 2*: *L1* is not empty.
  L1 has both a `first` component and a `rest` component. There are two cases: either `(first` *`L1`*`)` is a member of *L2* or it is not.
  - *Case 2.1*: `(first` *`L1`*`)` is a member of *L2*.
    `(first` *`L1`*`)` belongs to both *L1* and *L2*, and thus belong to their intersection. Therefore, the intersection of *L1* and *L2* is simply `(first` *`L1`*`)` plus the intersection of `(rest` *`L1`*`)` and *L2*.

- *Case 2.2*: `(first L1)` is not a member of *L2*.
  Since `(first L1)` does not belong to *L2*, it does not belong to the intersection of *L1* and *L2*. As a result, the intersection of *L1* and *L2* is exactly the intersection of `(rest L1)` and *L2*.

A trace of executing the function is given below:

```
>(trace list-intersection)
(LIST-INTERSECTION)
> (list-intersection '(1 3 5 7) '(1 2 3 4))
 0: (LIST-INTERSECTION (1 3 5 7) (1 2 3 4))
   1: (LIST-INTERSECTION (3 5 7) (1 2 3 4))
     2: (LIST-INTERSECTION (5 7) (1 2 3 4))
       3: (LIST-INTERSECTION (7) (1 2 3 4))
         4: (LIST-INTERSECTION NIL (1 2 3 4))
         4: returned NIL
       3: returned NIL
     2: returned NIL
   1: returned (3)
 0: returned (1 3)
(1 3)
```

Vectors are one-dimensional arrays, therefore a subtype of array. Vectors and lists are collectively called sequences. Therefore all sequence generic functions and array functions we have discussed so far, work on vectors.

8. Creating Vectors

The vector function allows you to make fixed-size vectors with specific values. It takes any number of arguments and returns a vector containing those arguments.

Example 1

Create a new source code file named main.lisp and type the following code in it.

```
(setf v1 (vector 1 2 3 4 5))
(setf v2 #(a b c d e))
(setf v3 (vector 'p 'q 'r 's 't))

(write v1)
(terpri)
(write v2)
(terpri)
(write v3)
```

When you execute the code, it returns the following result −

```
#(1 2 3 4 5)
#(A B C D E)
```

```
#(P Q R S T)
```

Please note that LISP uses the #(...) syntax as the literal notation for vectors. You can use this #(...) syntax to create and include literal vectors in your code.

However, these are literal vectors, so modifying them is not defined in LISP. Therefore, for programming, you should always use the **vector** function, or the more general function **make-array** to create vectors you plan to modify.

The **make-array** function is the more generic way to create a vector. You can access the vector elements using the **aref** function.

Example 2

Create a new source code file named main.lisp and type the following code in it.

```
(setq a (make-array 5 :initial-element 0))
(setq b (make-array 5 :initial-element 2))

(dotimes (i 5)
   (setf (aref a i) i))

(write a)
(terpri)
(write b)
(terpri)
```

When you execute the code, it returns the following result −

```
#(0 1 2 3 4)
#(2 2 2 2 2)
```

Exercise:

1. Explain the structure of for – loop in LISP.
2. Write a program to create factorial function.
3. Explain difference between macro and function.
4. Write a lisp program to check an even number.
5. Explain *member* and nth keywords.
6. What is List? Explain with suitable example.

Chapter 5

Functional Programming in LISP

1. LISP – SET

Implementing Sets in LISP

Sets, like lists are generally implemented in terms of cons cells. However, for this very reason, the set operations get less and less efficient the bigger the sets get.

The **adjoin** function allows you to build up a set. It takes an item and a list representing a set and returns a list representing the set containing the item and all the items in the original set.

The **adjoin** function first looks for the item in the given list, if it is found, then it returns the original list; otherwise it creates a new cons cell with its **car** as the item and **cdr** pointing to the original list and returns this new list.

The **adjoin** function also takes **:key** and **:test** keyword arguments. These arguments are used for checking whether the item is present in the original list.

Since, the adjoin function does not modify the original list, to make a change in the list itself, you must either assign the value returned by adjoin to the original list or, you may use the macro **pushnew** to add an item to the set.

Example

Create a new source code file named main.lisp and type the following code in it.

```
; creating mysetas an empty list
(defparameter*myset*())
(adjoin1*myset*)
(adjoin2*myset*)

; adjoin did not change the original set
;so it remains same
(write*myset*)
(terpri)
(setf*myset*(adjoin 1*myset*))
(setf*myset*(adjoin 2*myset*))

;now the original setis changed
(write*myset*)
(terpri)
```

```
;adding an existing value
(pushnew2*myset*)

;no duplicate allowed
(write*myset*)
(terpri)

;pushing a new value
(pushnew3*myset*)
(write*myset*)
(terpri)
```

When you execute the code, it returns the following result −

```
NIL
(2 1)
(2 1)
(3 2 1)
```

Checking Membership

The member group of functions allows you to check whether an element is member of a set or not.

The following are the syntaxes of these functions −

```
member item list &key :test :test-not :key
member-if predicate list &key :key
member-if-not predicate list &key :key
```

These functions search the given list for a given item that satisfies the test. If no such item is found, then the functions returns **nil.** Otherwise, the tail of the list with the element as the first element is returned.

The search is conducted at the top level only.

These functions could be used as predicates.

Example

Create a new source code file named main.lisp and type the following code in it.

```
(write(member 'zara '(ayanabdulzarariyannuha)))
(terpri)
(write(member-if#'evenp '(3 7 2 5/3 'a)))
(terpri)
(write(member-if-not#'numberp '(3 7 2 5/3 'a 'b 'c)))
```

When you execute the code, it returns the following result −

```
(ZARA RIYAN NUHA)
(2 5/3 'A)
('A 'B 'C)
```

Set Union

The union group of functions allows you to perform set union on two lists provided as arguments to these functions on the basis of a test.

The following are the syntaxes of these functions −

```
union list1 list2 &key :test :test-not :key
nunion list1 list2 &key :test :test-not :key
```

The **union** function takes two lists and returns a new list containing all the elements present in either of the lists. If there are duplications, then only one copy of the member is retained in the returned list.

The **nunion** function performs the same operation but may destroy the argument lists.

Example

Create a new source code file named main.lisp and type the following code in it.

```
(setq set1 (union'(a b c) '(c d e)))
(setq set2 (union'(#(a b) #(5 6 7) #(f h))
   '(#(567)#(a b) #(g h)) :test-not #'mismatch)
)

(setq set3 (union'(#(a b) #(5 6 7) #(f h))
   '(#(567)#(a b) #(g h)))
)
(write set1)
(terpri)
(write set2)
(terpri)
(write set3)
```

When you execute the code, it returns the following result −

```
(A B C D E)
(#(F H) #(5 6 7) #(A B) #(G H))
(#(A B) #(5 6 7) #(F H) #(5 6 7) #(A B) #(G H))
```

Please Note

The union function does not work as expected without **:test-not #'mismatch** arguments for a list of three vectors. This is because, the lists are made of cons cells and although the values look same to us apparently, the **cdr** part of cells does not match, so they are not exactly same to LISP

interpreter/compiler. This is the reason; implementing big sets are not advised using lists. It works fine for small sets though.

Set Intersection

The intersection group of functions allows you to perform intersection on two lists provided as arguments to these functions on the basis of a test.

The following are the syntaxes of these functions −

```
intersection list1 list2 &key :test :test-not :key
nintersection list1 list2 &key :test :test-not :key
```

These functions take two lists and return a new list containing all the elements present in both argument lists. If either list has duplicate entries, the redundant entries may or may not appear in the result.

Example

Create a new source code file named main.lisp and type the following code in it.

```
(setq set1 (intersection '(a b c) '(c d e)))
(setq set2 (intersection '(#(a b) #(5 6 7) #(f h))
   '(#(567)#(a b) #(g h)) :test-not #'mismatch)
)

(setq set3 (intersection '(#(a b) #(5 6 7) #(f h))
   '(#(567)#(a b) #(g h)))
)
(write set1)
(terpri)
(write set2)
(terpri)
(write set3)
```

When you execute the code, it returns the following result −

```
(C)
(#(A B) #(5 6 7))
NIL
```

The intersection function is the destructive version of intersection, i.e., it may destroy the original lists.

Set Difference

The set-difference group of functions allows you to perform set difference on two lists provided as arguments to these functions on the basis of a test.

The following are the syntaxes of these functions −

```
set-difference list1 list2 &key :test :test-not :key
nset-difference list1 list2 &key :test :test-not :key
```

The set-difference function returns a list of elements of the first list that do not appear in the second list.

Example

Create a new source code file named main.lisp and type the following code in it.

```
setq set1 (set-difference '(a b c) '(c d e)))
(setq set2 (set-difference '(#(a b) #(5 6 7) #(f h))
   '(#(567)#(a b) #(g h)) :test-not #'mismatch)
)
(setq set3 (set-difference '(#(a b) #(5 6 7) #(f h))
   '(#(567)#(a b) #(g h)))
)
(write set1)
(terpri)
(write set2)
(terpri)
(write set3)
```

When you execute the code, it returns the following result −

```
(A B)
(#(F H))
(#(A B) #(5 6 7) #(F H))
```

2. Auxiliary Functions and Accumulator Variables

We define the *reversal* of a list *L* to be a list containing exactly the elements of *L* in reversed order. The Common LISP built-in function `(reverse L)` returns the reversal of *L*:

```
USER(1): (reverse '(1 2 3 4))
(4 3 2 1)
USER(2): (reverse '(1 (a b) (c d) 4))
(4 (c d) (a b) 1)
USER(3): (reverse nil)
NIL
```

Implementing a version of `reverse` is not difficult, but finding an efficient implementation is not as trivial.

```
(defun slow-list-reverse (L)
  "Create a new list containing the elements of L in reversed order."
  (if (null L)
nil
    (list-append (slow-list-reverse (rest L))
```

```
                (list (first L)))))
```

Notice that this linearly recursive implementation calls the function `list-append` we implemented in the the last tutorial. Notice also the new function `list`, which returns a list containing its arguments. For example, `(list 1 2)` returns the list `(1 2)`. In general, `(list` x_1 x_2 `...` x_n`)` is just a shorthand for `(cons` x_1 `(cons` x_2 `...` `(cons` x_n `nil) ... ))`. So, the expression `(list (first L))` in the listing above returns a singleton list containing the first element of `L`.

So, why does `(slow-list-reverse` *L*`)` returns the reversal of *L*? The list *L* is constructed either by `nil` or by `cons`:

- **Case 1:** *L* is `nil`.
  The reversal of *L* is simply `nil`.
- **Case 2:** *L* is constructed from a call to `cons`. Then `L` has two components: `(first` *L*`)` and `(rest` *L*`)`. If we append `(first` *L*`)` to the end of the reversal of `(rest` *L*`)`, then we obtain the reversal of *L*. Of course, we could make use of `list-append` to do this. However, `list-append` expects two list arguments, so we need to construct a singleton list containing `(first` *L*`)` before we pass it as a second argument to `list-append`.

Let us trace the execution of the function to see how the recursive calls unfold:

```
USER(3): (trace slow-list-reverse)
(SLOW-LIST-REVERSE)
USER(4): (slow-list-reverse '(1 2 3 4))
 0: (SLOW-LIST-REVERSE (1 2 3 4))
   1: (SLOW-LIST-REVERSE (2 3 4))
     2: (SLOW-LIST-REVERSE (3 4))
       3: (SLOW-LIST-REVERSE (4))
         4: (SLOW-LIST-REVERSE NIL)
         4: returned NIL
       3: returned (4)
     2: returned (4 3)
   1: returned (4 3 2)
 0: returned (4 3 2 1)
(4 3 2 1)
```

Everything looks fine, until we trace also the unfolding of `list-append`:

```
USER(9): (trace list-append)
(LIST-APPEND)
USER(10): (slow-list-reverse '(1 2 3 4))
 0: (SLOW-LIST-REVERSE (1 2 3 4))
   1: (SLOW-LIST-REVERSE (2 3 4))
     2: (SLOW-LIST-REVERSE (3 4))
       3: (SLOW-LIST-REVERSE (4))
         4: (SLOW-LIST-REVERSE NIL)
```

```
          4: returned NIL
          4: (LIST-APPEND NIL (4))
          4: returned (4)
        3: returned (4)
        3: (LIST-APPEND (4) (3))
          4: (LIST-APPEND NIL (3))
          4: returned (3)
        3: returned (4 3)
      2: returned (4 3)
      2: (LIST-APPEND (4 3) (2))
        3: (LIST-APPEND (3) (2))
          4: (LIST-APPEND NIL (2))
          4: returned (2)
        3: returned (3 2)
      2: returned (4 3 2)
    1: returned (4 3 2)
    1: (LIST-APPEND (4 3 2) (1))
      2: (LIST-APPEND (3 2) (1))
        3: (LIST-APPEND (2) (1))
          4: (LIST-APPEND NIL (1))
          4: returned (1)
        3: returned (2 1)
      2: returned (3 2 1)
    1: returned (4 3 2 1)
 0: returned (4 3 2 1)
(4 3 2 1)
```

What we see here is revealing: given a list of *N* element, `slow-list-reverse` makes *O(N)* recursive calls, with each level of recursion l involving a call to the linear-time function `list-append`. The result is that `slow-list-reverse` is an $O(N^2)$ function.

We can in fact build a much more efficient version of `reverse` using *auxiliary functions* and *accumulator variables*:

```
(defun list-reverse (L)
  "Create a new list containing the elements of L in reversed order."
  (list-reverse-aux L nil))

(defun list-reverse-aux (L A)
  "Append list A to the reversal of list L."
  (if (null L)
      A
    (list-reverse-aux (rest L) (cons (first L) A))))
```

The function `list-reverse-aux` is an *auxiliary function* (or a *helper function*). It does not perform any useful function by itself, but the *driver function* `list-reverse` could use it as a tool when building a reversal. Specifically, `(list-reverse-aux L A)` returns a new list obtained by appending list *A* to the reversal of list *L*. Bypassing `nil` as *A* to `list-reverse-aux`, the driver function `list-reverse` obtains the reversal of *L*.

Let us articulate why `(list-reverse-aux L A)` correctly appends *A* to the reversal of list *L*. Again, we know that either *L* is `nil` or it is constructed by `cons`:

- **Case 1:** *L* is `nil`.
  The reversal of *L* is simply `nil`. The result of appending *A* to the end of an empty list is simply *A* itself.
- **Case 2:** *L* is constructed by `cons`.
  Now *L* is composed of two parts: `(first L)` and `(rest L)`. Observe that `(first L)` is the last element in the reversal of *L*. If we are to append *A* to the end of the reversal of *L*, then `(first L)` will come immediately before the elements of *A*. Observing the above, we recognize that we obtain the desired result by recursively appending `(cons (first L) A)` to the reversal of `(rest L)`.

3. Functions as First-Class Objects

A data type is *first-class* in a programming language when you can pass instances of the data type as function arguments or return them as function values. We are used to treating numeric and Boolean values as first-class data types in languages like Pascal and C. However, we might not be familiar to the notion that functions could be treated as first-class objects, that is, functions can be passed as function arguments and returned as function values. This unique feature of Common LISP and other functional languages makes it easy to build very powerful abstractions. In the remaining of this tutorial, we will look at what passing functional arguments buys us. In the fourth tutorial, when we talk about imperative programming in LISP, we will return to the topic of returning functional values.

Frequently, we need to apply a transformation multiple times on the same data object. For example, we could define the following transformation:

```
(defun double (x)
  "Multiple X by 2."
  (* 2 x))
```

We could compute 2^4 by applying the `double` transformation 4 times on 1:

```
USER(10): (double (double (double (double 1))))
16
```

Not only is this clumsy, but it also fails to express the very idea that the same transformation is applied multiple times. It would be nice if we can repeat applying a given transformation *F* on *X*for *N* times by simply writing `(repeat-transformation F N X)`. The following function does just that:

```
(defun repeat-transformation (F N X)
  "Repeat applying function F on object X for N times."
  (if (zerop N)
```

```
      X
    (repeat-transformation F (1- N) (funcall F X))))
```

The definition follows the standard tail recursive pattern. Notice theform `(funcall F X)`. Given a function *F* and objects X_1 X_2 ... X_n, the form `(funcall F` X_1 X_2 `...` X_n`)` invoke the function *F*with arguments X_1, X_2, ..., X_n. The variable *N* is a counter keeping track of the remaining number of times we need to apply function *F* to the accumulator variable *X*.

To pass a the function `double` as an argument to `repeat-transformation`, we need to annotate the function name `double` by a *closure constructor*, as in the following:

```
USER(11): (repeat-transformation (function double) 4 1)
16
```

There is nothing magical going on, the closure constructor is just a syntax for telling Common LISP that what follows is a function rather than a local variable name. Had we not included the annotation, Common LISP will treat the name `double` as a variable name, and then report an error since the name `double` is not defined.

To see how the evaluation arrives at the result 16, we could, as usual, trace the execution:

```
USER(12): (trace repeat-transformation)
REPEAT-TRANSFORMATION
USER(13): (repeat-transformation #'double 4 1)
 0: (REPEAT-TRANSFORMATION # 4 1)
   1: (REPEAT-TRANSFORMATION # 3 2)
     2: (REPEAT-TRANSFORMATION # 2 4)
       3: (REPEAT-TRANSFORMATION # 1 8)
         4: (REPEAT-TRANSFORMATION # 0 16)
         4: returned 16
       3: returned 16
     2: returned 16
   1: returned 16
 0: returned 16
16
```

4. Higher-Order Functions

Notice that exponentiation is not the only use of the `repeat-transformation` function. Let's say we want to build a list containing 10 occurrences of the symbol `blah`. We can do so with the help of `repeat-transformation`:

```
USER(30): (defun prepend-blah (L) (cons 'blah L))
PREPEND-BLAH
USER(31): (repeat-transformation (function prepend-blah) 10 nil)
(BLAH BLAHBLAHBLAHBLAHBLAHBLAHBLAHBLAH BLAH)
```

Suppose we want to fetch the 7'th element of the list `(a b c d e f g h i j)`. Of course, we could use the built in function `seventh` to do the job, but for the fun of it, we could also achieve what we want in the following way:

```
USER(32): (first (repeat-transformation (function rest) 6 '(a b c d e f g h i
j)))
G
```

Basically, we apply `rest` six times before apply `first` to get the seventh element. In fact, we could have defined the function `list-nth` (see previous tutorial) in the following way:

```
(defun list-nth (N L)
  (first (repeat-transformation (function rest) N L)))
```

(`list-nth` numbers the member of a list from zero onwards.)

As you can see, functions that accepts other functions as arguments are very powerful abstractions. You can encapsulate generic algorithms in such a function, and parameterize their behavior by passing in different function arguments. We call a function that has functional parameters (or return a function as its value) a *higher-order* function.

One last point before we move on. The closure constructor `function` is used very often when working with higher-order functions. Common LISP therefore provide another equivalent syntax to reduce typing. When we want Common LISP to interpret a name `F` as a function, instead of typing `(function F)`, we can also type the shorthand `#'F`. The prefix `#'` is nothing but an alternative syntax for the closure constructor. For example, we could enter the following:

```
USER(33): (repeat-transformation #'double 4 1)
16
USER(34): (repeat-transformation #'prepend-blah 10 nil)
(BLAH BLAHBLAHBLAHBLAHBLAHBLAHBLAHBLAH BLAH)
USER(35): (first (repeat-transformation #'rest 6 '(a b c d e f g h i j)))
G
```

5. Lambda Expressions

Some of the functions, like `prepend-blah` for example, serves no other purpose but to instantiate the generic algorithm `repeat-transformation`. It would be tedious if we need to define it as a global function using `defun` before we pass it into `repeat-transformation`. Fortunately, LISP provides a mechanism to help us define functions "in place":

```
USER(36): (repeat-transformation #'(lambda (L) (cons 'blah L)) 10 nil)
(BLAH BLAHBLAHBLAHBLAHBLAHBLAHBLAHBLAH BLAH)
```

The first argument `(lambda (L) (cons 'blah L))` is a *lambda expression*. It designates an anonymous function (nameless function) with one parameter `L`, and it returns as a function value `(cons 'blah L)`. We prefix the lambda expression with the closure constructor `#'` since we want Common LISP to interpret the argument as a function rather than a call to a function named `lambda`.

Similarly, we could have computed powers as follows:

```
USER(36): (repeat-transformation #'(lambda (x) (* 2 x)) 4 1)
16
```

6. Iterating Through a List

We have seen how we could iterate through a list using linear recursion. We have also seen how such recursion can be optimized if structured in a tail-recursive form. However, many of the the list processing functions look striking similar. Consider the following examples:

```
(defun double-list-elements (L)
  "Given a list L of numbers, return a list containing the elements of L
multiplied by 2."
  (if (null L)
nil
    (cons (double (first L)) (double-list-elements (rest L)))))

(defun reverse-list-elements (L)
  "Given a list L of lists, return a list containing the reversal of L's
members."
  (if (null L)
nil
    (cons (reverse (first L)) (reverse-list-elements (rest L)))))
```

We could come up with infinitely many more examples of this sort. Having to write a new function everytime we want to iterate through a list is both time-consuming and error-prone. The solution is to capture the generic logic of list iteration in higher-order functions, and specialize their behaviors by passing in functional arguments.

If we look at the definitions of `double-list-elements` and `reverse-list-elements`, we see that they apply a certain function to the `first` element of their input, then recursively invoke themselves to process the `rest` of the input list, and lastly combine the result of the two function calls using `cons`. We could capture this logic into the following function:

```
(defunmapfirst (F L)
  "Apply function F to every element of list L, and return a list containing
the results."
  (if (null L)
nil
    (cons (funcall F (first L)) (mapfirst F (rest L)))))
```

The functions `double-list-elements` and `reverse-list-elements` can be replaced by the following:

```
USER(18): (mapfirst #'double '(1 2 3 4))
(2 4 6 8)
USER(19): (mapfirst #'reverse '((1 2 3) (a b c) (4 5 6) (d e f)))
((3 2 1) (C B A) (6 5 4) (F E D))
```

Of course, you could also pass lambda abstractions as arguments:

```
USER(20): (mapfirst #'(lambda (x) (* x x)) '(1 2 3 4))
(1 4 9 16)
```

In fact, the higher-order function is so useful that Common LISP defines a function `mapcar` that does exactly what `mapfirst` is intended for:

```
USER(22): (mapcar #'butlast '((1 2 3) (a b c) (4 5 6) (d e f)))
((1 2) (A B) (4 5) (D E))
```

The reason why it is called `mapcar` is that the function `first` was called `car` in some older dialects of LISP (and `rest` was called `cdr` in those dialects; Common LISP still supports `car` and `cdr` but we strongly advice you to stick with the more readable `first` and `rest`). We suggest you to consider using `mapcar` whenever you are tempted to write your own list-iterating functions.

The function `mapcar` is an example of *generic iterators*, which capture the generic logic of iterating through a list. If we look at what we do the most when we iterate through a list, we find that the following kinds of iterations occurs most frequently in our LISP programs:

1. *Transformation iteration*: transforming a list by systematically applying a monadic function to the elements of the list.
2. *Search iteration*: searching for a list member that satisfies a given condition.
3. *Filter iteration*: screening out all members that does not satisfy a given condition.

As we have already seen, `mapcar` implements the generic algorithm for performing transformation iteration. In the following, we will look at the analogous of `mapcar` for the remaining iteration categories.

7. Search Iteration

Let us begin by writing a function that returns an even element in a list of numbers:

```
(defun find-even (L)
  "Given a list L of numbers, return the leftmost even member."
```

```
  (if (null L)
nil
    (if (evenp (first L))
        (first L)
      (find-even (rest L)))))
```

We notice that the essential logic of searching can be extracted out into the following definition:

```
(defun list-find-if (P L)
  "Find the leftmost element of list L that satisfies predicate P."
  (if (null L)
nil
    (if (funcall P (first L))
        (first L)
      (list-find-if P (rest L)))))
```

The function `list-find-if` examines the elements of *L* one by one, and return the first one that satisfies predicate *P*. The function can be used for locating even or non-`nil` members in a list:

```
USER(34): (list-find-if #'evenp '(1 3 5 8 11 12))
8
USER(35): (list-find-if #'(lambda (X) (not (null X))) '(nil nil (1 2 3) (4
5)))
(1 2 3)
```

Common LISP defines a built-in function `find-if` which is a more general version of `list-find-if`. It can be used just like `list-find-if`:

```
USER(37): (find-if #'evenp '(1 3 5 8 11 12))
8
USER(38): (find-if #'(lambda (X) (not (null X))) '(nil nil (1 2 3) (4 5)))
(1 2 3)
```

8. Functions Returning Multiple Values

The functions we have seen so far return a single value, but some LISP functions return two or more values. For example, if two arguments *number* and *divisor* are passed to the `floor` function, it returns two values, the quotient *q* and the remainder *r* so that *number = divisor * q + r*.

```
USER(11): (floor 17 5)
3
2
USER(12): (floor -17 5)
-4
3
```

Regular binding constructs like `let` and `let*` only allows us to catch the first returned value of a multiple-valued function, as the following example illustrates:

```
USER(13): (let ((x (floor 17 5))) x)
3
```

One can use the `multiple-value-bind` to receive the results of a multiple-valued function:

```
USER(14): (multiple-value-bind (x y) (floor 17 5)
            (+ x y))
5
```

In the above expression, `(x y)` is the list of variables binding to the returned values, `(floor 17 5)` is the expression generating multiple results. Binding the two values of `(floor 17 5)` to `x` and `y`, LISP then evaluates the expression `(+ x y)`.

One may also write LISP functions that return multiple values:

```
(defun order (a b)
  "Return two values: (min a b) and (max a b)."
  (if (>= a b)
      (values b a)
    (values a b)))
```

The `values` special form returns its arguments as multiple values.

Exercise:

1. Explain SET operations with suitable example.
2. Explain Union with suitable example.
3. Explain Set Difference with suitable example.
4. What is intersection? Explain how it is solved.
5. Explain high order functions.
6. Explain in brief lambda expressions.
7. Give example for Search iterations.

Chapter 6
Advanced LISP

1. Data Abstraction

Binary Trees

Suppose we want to create a new kind of *recursive data type*, our familiar binary trees. The first thing we have to do is to define the data type in terms of its *constructors*, *selectors* and *recognizers*. In the case of binary trees, we have the following:

1. *Constructors*: We have two kinds of binary trees, *leaves* and *nodes*. Accordingly, we need a constructor for each kind:
   - `(make-bin-tree-leaf E)`: A leaf is a composite object with one component, the *element E*.
   - `(make-bin-tree-node E B1 B2)`: A node consists of three components, an element *E*, a *left subtree B1* and a *right subtree B2*. Each of *B1* and *B2* is a binary tree.

   Notice that the definition of binary tree is inherently recursive (as in the case of nodes). Larger binary trees can be composed from smaller ones.

2. *Selectors*: We need to define a selector for each component of each kind of binary tree.
   - `(bin-tree-leaf-element L)`: Retrieve the element of a leaf *L*.
   - `(bin-tree-node-element N)`: Retrieve the element of a node *N*.
   - `(bin-tree-node-left N)`: Retrieve the left subtree of a node *N*.
   - `(bin-tree-node-right N)`: Retrieve the right subtree of a node *N*.
3. *Recognizers*: We define one recognizer for each kind of binary tree.
   - `(bin-tree-leaf-p B)`: Test if a given binary tree *B* is a leaf.
   - `(bin-tree-node-p B)`: Test if a given binary tree *B* is a node.

Notice that we have not written a line of code yet, and still we are able to write down the function signature of all the constructors, selectors and recognizers. The process is more or less mechanical:

1. Define a constructor for each variant of the recursive data type. The parameters for a constructor defines the components of a composite object.
2. For each parameter of each constructor, define a selector to retrieve the corresponding component.
3. For each constructor, define a corresponding recognizer.

The next question is how we are to *represent* a binary tree as a LISP object. Of course, a list is the first thing that comes to our mind:

- We represent an leaf with element *E* by a singleton list containing *E* (i.e. `(list E)`).
- A node with element *E*, left subtree *B1* and right subtree *B2* is represented as a list containing the three components (i.e. `(list E B1 B2)`).

Fixing the representation, we can thus implement the recursive data type functions:

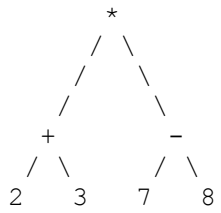
```
(defun make-bin-tree-leaf (E)
  "Create a leaf."
  (list E))

(defun make-bin-tree-node (E B1 B2)
  "Create a node with element K, left subtree B1 and right subtree B2."
  (list E B1 B2))

;;
;; Selectors for binary trees
;;

(defun bin-tree-leaf-element (L)
  "Retrieve the element of a leaf L."
  (first L))

(defun bin-tree-node-element (N)
  "Retrieve the element of a node N."
  (first N))

(defun bin-tree-node-left (N)
  "Retrieve the left subtree of a node N."
  (second N))

(defun bin-tree-node-right (N)
  "Retrieve the right subtree of a node N."
  (third N))

;;
;; Recognizers for binary trees
;;

(defun bin-tree-leaf-p (B)
  "Test if binary tree B is a leaf."
  (and (listp B) (= (list-length B) 1)))

(defun bin-tree-node-p (B)
  "Test if binary tree B is a node."
  (and (listp B) (= (list-length B) 3)))
```

The representation scheme works out like the following:

```
USER(5): (make-bin-tree-node '*
                             (make-bin-tree-node '+
                                                 (make-bin-tree-leaf 2)
                                                 (make-bin-tree-leaf 3))
```

```
                                (make-bin-tree-node '-
                                                    (make-bin-tree-leaf 7)
                                                    (make-bin-tree-leaf 8)))
(* (+ (2) (3)) (- (7) (8)))
```

The expression above is a binary tree node with element `*` and two subtrees. The left subtree is itself a binary tree node with `+` as its element and leaves as its subtress. The right subtree is also a binary tree node with `-` as its element and leaves as its subtrees. All the leaves are decorated by numeric components.



Searching Binary Trees

As discussed in previous tutorials, having recursive data structures defined in the way we did streamlines the process of formulating structural recursions. We review this concept in the following examples.

Suppose we treat binary trees as containers. An expression *E* is a member of a binary tree *B* if:

1. *B* is a leaf and its element is *E*.
2. *B* is a node and either its element is *E* or *E* is a member of one of its subtrees.

For example, the definition asserts that the members of `(* (+ (2) (3)) (- (7) (8)))` are `*`, `+`, `2`, `3`, `-`, `7` and `8`. Such a definition can be directly implemented by our recursive data type functions:

```
(defun bin-tree-member-p (B E)
  "Test if E is an element in binary tree B."
  (if (bin-tree-leaf-p B)
      (equal E (bin-tree-leaf-element B))
    (or (equal E (bin-tree-node-element B))
        (bin-tree-member-p (bin-tree-node-left B) E)
        (bin-tree-member-p (bin-tree-node-right B) E))))
```

The function can be made more readable by using the `let` form:

```
(defun bin-tree-member-p (B E)
  "Test if E is an element in binary tree B."
  (if (bin-tree-leaf-p B)
      (equal E (bin-tree-leaf-element B))
    (let
        ((elmt  (bin-tree-node-element B))
         (left  (bin-tree-node-left    B))
         (right (bin-tree-node-right   B)))
      (or (equal E elmt)
          (bin-tree-member-p left E)
          (bin-tree-member-p right E)))))
```

Tracing the execution of `bin-tree-member-p`, we get:

```
USER(14): (trace bin-tree-member-p)
(BIN-TREE-MEMBER-P)
USER(15): (bin-tree-member-p '(+ (* (2) (3)) (- (7) (8))) 7)
 0: (BIN-TREE-MEMBER-P (+ (* (2) (3)) (- (7) (8))) 7)
   1: (BIN-TREE-MEMBER-P (* (2) (3)) 7)
     2: (BIN-TREE-MEMBER-P (2) 7)
     2: returned NIL
     2: (BIN-TREE-MEMBER-P (3) 7)
     2: returned NIL
   1: returned NIL
   1: (BIN-TREE-MEMBER-P (- (7) (8)) 7)
     2: (BIN-TREE-MEMBER-P (7) 7)
     2: returned T
   1: returned T
 0: returned T
T
```

2. Abstract Data Types

Abstract data types are blackboxes. They are defined in terms of their external interfaces, and not their implementation. For example, a *set* abstraction offers the following operations:

- `(make-empty-set)` creates an empty set.
- `(set-insert S E)` returns a set containing all members of set *S* plus an additional member *E*.
- `(set-remove S E)` returns a set containing all members of set *S* except for *E*.
- `(set-member-p S E)` returns true if *E* is a member of set *S*.
- `(set-empty-p S)` returns true if set *S* is empty.

To implement an abstract data type, we need to decide on a representation. Let us represent a set by a list with no repeated members.

```
(defun make-empty-set ()
  "Creates an empty set."
  nil)

(defun set-insert (S E)
  "Return a set containing all the members of set S plus the element E."
  (adjoin E S :test #'equal))

(defun set-remove (S E)
  "Return a set containing all the members of set S except for element E."
  (remove E S :test #'equal))

(defun set-member-p (S E)
  "Return non-NIL if set S contains element E."
  (member E S :test #'equal))

(defun set-empty-p (S)
```

```
  "Return true if set S is empty."
  (null S))
```

Notice that we have implemented an abstract data type (sets) using a more fundamental recursive data structure (lists) with additional computational constraints (no repetition) imposed by the interface functions.

3. Tower of Hanoi

The Tower of Hanoi problem is a classical toy problem in Artificial Intelligence: There are N disks D_1, D_2, ..., D_n, of graduated sizes and three pegs 1, 2, and 3. Initially all the disks are stacked on peg 1, with D_1, the smallest, on top and D_n, the largest, at the bottom. The problem is to transfer the stack to peg 3 given that only one disk can be moved at a time and that no disk may be placed on top of a smaller one.

We call peg 1 the "from" peg, peg 3 the "to" peg. Peg 2 is a actually a buffer to facilitate movement of disks, and we call it an "auxiliary" peg. We can move N disks from the "from" peg to the "to" peg using the following recursive scheme.

1. *Ignoring the largest disk at the "from" peg, treat the remaining disks as a Tower of Hanoi problem with N-1 disks. Recursively move the top N-1 disks from the "from" peg to the "auxiliary" peg, using the "to" peg as a buffer.*
2. *Now that the N-1 smaller disks are in the "auxiliary" peg, we move the largest disk to the "to" peg.*
3. *Ignoring the largest disk again, treat the remaining disks as a Tower of Hanoi problem with N-1 disks. Recursively move the N-1 disks from the "auxiliary" peg to the "to" peg, using the "from" peg as a buffer.*

To code this solution in LISP, we need to define some data structure. First, we represent a disk by a number, so that D_i is represented by i. Second, we represent a stack of disks by a tower, which is nothing but a list of numbers, with the first element representing the top disk. We define the usual constructors and selectors for the tower data type.

```
(defun make-empty-tower ()
  "Create tower with no disk."
  nil)

(defun tower-push (tower disk)
  "Create tower by stacking DISK on top of TOWER."
  (cons disk tower))

(defun tower-top (tower)
  "Get the top disk of TOWER."
  (first tower))

(defun tower-pop (tower)
  "Remove the top disk of TOWER."
```

```
  (rest tower))
```

Third, we define the hanoi data type to represent a Tower of Hanoi configuration. In particular, a hanoi configuration is a list of three towers. The elementary constructors and selectors are given below:

```
(defun make-hanoi (from-tower aux-tower to-tower)
  "Create a Hanoi configuration from three towers."
  (list from-tower aux-tower to-tower))

(defunhanoi-tower (hanoii)
  "Select the I'th tower of a Hanoi construction."
  (nth (1- i) hanoi))
```

Working with towers within a Hanoi configuration is tedious. We therefore define some shortcut to capture recurring operations:

```
(defunhanoi-tower-update (hanoii tower)
  "Replace the I'th tower in the HANOI configuration by tower TOWER."
  (cond
   ((= i 1) (make-hanoi tower (second hanoi) (third hanoi)))
   ((= i 2) (make-hanoi (first hanoi) tower (third hanoi)))
   ((= i 3) (make-hanoi (first hanoi) (second hanoi) tower))))

(defunhanoi-tower-top (hanoii)
  "Return the top disk of the I'th tower in the HANOI configuration."
  (tower-top (hanoi-tower hanoii)))

(defunhanoi-tower-pop (hanoii)
  "Pop the top disk of the I'th tower in the HANOI configuration."
  (hanoi-tower-update hanoii (tower-pop (hanoi-tower hanoii))))

(defunhanoi-tower-push (hanoii disk)
  "Push DISK into the I'th tower of the HANOI configuration."
  (hanoi-tower-update hanoii (tower-push (hanoi-tower hanoii) disk)))
```

The fundamental operator we can perform on a Hanoi configuration is to move a top disk from one peg to another:

```
(defun move-disk (from to hanoi)
  "Move the top disk from peg FROM to peg TO in configuration HANOI."
  (let
      ((disk               (hanoi-tower-top hanoi from))
       (intermediate-hanoi (hanoi-tower-pop hanoi from)))
    (hanoi-tower-push intermediate-hanoi to disk)))
```

We are now ready to capture the logic of our recursive solution into the following code:

```
(defun move-tower (N from aux to hanoi)
  "In the HANOI configuration, move the top N disks from peg FROM to peg TO
using peg AUX as an auxiliary peg."
  (if (= N 1)
```

```
      (move-disk from to hanoi)
    (move-tower (- N 1) aux from to
                (move-disk from to
                           (move-tower (- N 1) from to aux hanoi)))))
```

We use the driver function `solve-hanoi` *to start up the recursion:*

```
(defun solve-hanoi (N)
  "Solve the Tower of Hanoi problem."
  (move-tower N 1 2 3 (make-hanoi (make-complete-tower N) nil nil)))

(defun make-complete-tower (N)
  "Create a tower of N disks."
  (make-complete-tower-aux N (make-empty-tower)))

(defun make-complete-tower-aux (N A)
  "Push a complete tower of N disks on top of tower A."
  (if (zerop N)
      A
    (make-complete-tower-aux (1- N) (tower-push A N))))
```

To solve a Tower of Hanoi problem with 3 disks, we call `(solve-hanoi 3)`*:*

```
USER(50): (solve-hanoi 3)
(NIL NIL (1 2 3))
```

All we get back is the final configuration, which is not as interesting as knowing the sequence of moves taken by the algorithm. So we trace usage of the `move-disk` *operator:*

```
USER(51): (trace move-disk)
(MOVE-DISK)
USER(52): (solve-hanoi 3)
 0: (MOVE-DISK 1 3 ((1 2 3) NIL NIL))
 0: returned ((2 3) NIL (1))
 0: (MOVE-DISK 1 2 ((2 3) NIL (1)))
 0: returned ((3) (2) (1))
 0: (MOVE-DISK 3 2 ((3) (2) (1)))
 0: returned ((3) (1 2) NIL)
 0: (MOVE-DISK 1 3 ((3) (1 2) NIL))
 0: returned (NIL (1 2) (3))
 0: (MOVE-DISK 2 1 (NIL (1 2) (3)))
 0: returned ((1) (2) (3))
 0: (MOVE-DISK 2 3 ((1) (2) (3)))
 0: returned ((1) NIL (2 3))
 0: (MOVE-DISK 1 3 ((1) NIL (2 3)))
 0: returned (NIL NIL (1 2 3))
(NIL NIL (1 2 3))
```

From the trace we can actually read off the sequence of operator applications necessary for one to achieve the solution configuration. This is good, but not good enough. We want to know why each move is being taken. So we trace also the high-level subgoals:

```
USER(53): (trace move-tower)
(MOVE-TOWER)
USER(54): (solve-hanoi 3)
 0: (MOVE-TOWER 3 1 2 3 ((1 2 3) NIL NIL))
```

```
   1: (MOVE-TOWER 2 1 3 2 ((1 2 3) NIL NIL))
     2: (MOVE-TOWER 1 1 2 3 ((1 2 3) NIL NIL))
       3: (MOVE-DISK 1 3 ((1 2 3) NIL NIL))
       3: returned ((2 3) NIL (1))
     2: returned ((2 3) NIL (1))
     2: (MOVE-DISK 1 2 ((2 3) NIL (1)))
     2: returned ((3) (2) (1))
     2: (MOVE-TOWER 1 3 1 2 ((3) (2) (1)))
       3: (MOVE-DISK 3 2 ((3) (2) (1)))
       3: returned ((3) (1 2) NIL)
     2: returned ((3) (1 2) NIL)
   1: returned ((3) (1 2) NIL)
   1: (MOVE-DISK 1 3 ((3) (1 2) NIL))
   1: returned (NIL (1 2) (3))
   1: (MOVE-TOWER 2 2 1 3 (NIL (1 2) (3)))
     2: (MOVE-TOWER 1 2 3 1 (NIL (1 2) (3)))
       3: (MOVE-DISK 2 1 (NIL (1 2) (3)))
       3: returned ((1) (2) (3))
     2: returned ((1) (2) (3))
     2: (MOVE-DISK 2 3 ((1) (2) (3)))
     2: returned ((1) NIL (2 3))
     2: (MOVE-TOWER 1 1 2 3 ((1) NIL (2 3)))
       3: (MOVE-DISK 1 3 ((1) NIL (2 3)))
       3: returned (NIL NIL (1 2 3))
     2: returned (NIL NIL (1 2 3))
   1: returned (NIL NIL (1 2 3))
 0: returned (NIL NIL (1 2 3))
(NIL NIL (1 2 3))
```

The trace gives us information as to what subgoals each operator application is trying to establish. For example, the top level subgoals are the following:

```
 0: (MOVE-TOWER 3 1 2 3 ((1 2 3) NIL NIL))
   1: (MOVE-TOWER 2 1 3 2 ((1 2 3) NIL NIL))
     ...
   1: returned ((3) (1 2) NIL)
   1: (MOVE-DISK 1 3 ((3) (1 2) NIL))
   1: returned (NIL (1 2) (3))
   1: (MOVE-TOWER 2 2 1 3 (NIL (1 2) (3)))
     ...
   1: returned (NIL NIL (1 2 3))
 0: returned (NIL NIL (1 2 3))
```

They translate directly to the following: In order to move a tower of 3 disks from peg 1 to peg 3 using peg 2 as a buffer (i.e. `(MOVE-TOWER 3 1 2 3 ((1 2 3) NIL NIL))`*) we do the following:*

1. `"1: (MOVE-TOWER 2 1 3 2 ((1 2 3) NIL NIL))"`
   Move a tower of 2 disks from peg 1 to peg 2 using peg 3 as a buffer. The result of the move is the following:
   `"1: returned ((3) (1 2) NIL)"`
2. `"1: (MOVE-DISK 1 3 ((3) (1 2) NIL))"`
   Move a top disk from peg 1 to peg 3. The result of this move is:
   `"1: returned (NIL (1 2) (3))"`
3. `"1: (MOVE-TOWER 2 2 1 3 (NIL (1 2) (3)))"`
   Move a tower of 2 disks from peg 2 to peg 3 using peg 1 as a buffer, yielding

the following configuration:

"1: returned (NIL NIL (1 2 3))"

4. The Eight Queens Puzzle

The rules are simple. You must place eight queens on a chessboard in such a manner that no queen is on either the same rank, file, or diagonal as another queen. In the terminology of cognitive psychologists, this is a very knowledge lean, well defined problem.

What makes it difficult is the size of the problem space. There are 64 possible places to place eight queens, a total of 4,426,165,368 possibilities.

An algorithm to find the solutions is simple, and relies on the following fact: since there are eight rows and eight columns, and also eight queens, each row and each column must contain one and only one queen. So, by placing a single queen on the b oard, and then another, and another, making sure each is legal, you can develop an algorithm to find any and all correct answers.
If you are doing this manually, you can just look to see. If you are letting a computer do the boring work (this is an OK but wimpy solution) you can have the chessboard be represented by an 8x8 array, and a queen will be illegally placed if it:
Has the same column number as another piece already placed
Has the same row number as another piece already placed
The slope between it and another placed piece is ± 1
So, since each column must have one queen, you can algorithmically choose a place for the queen in column 1, then column 2, etc. When you can not place the next queen, (when none of the spaces in its column are free) you can move back to the previous column and find the next square that works. This could probably be coded recursively in about 5 lines.
By using this algorithm, you will find all possible combinations that will work. Additionally, you only need to go through the first four rows of the first queen, because the last four will only give the solutions you arrived at initially, although perhaps reflected or reversed.

LISP solution to 8 queens problem:

```
;;;The following common lisp program is based on a solution given by
;;;Winston and Horn ("Lisp 3rd edition",1989, p.289, 538-539.)
;;;Significant improvements and alterations were made in all but the most
;;;primitive functions.  To run, simply load file and type (queen n) where
;;;n is the width of board.

;discovers if a piece threatens another
(defun threat (i j a b)
  (or (= i a)
      (= j b)
      (= (- i j) (- a b))
      (= (+ i j) (+ a b))))
```

```
;discovers if a placement is OK
(defun conflict (n m board)
  (cond  ((endp board) nil)
         ((threat n
                  m
                  (first (first board))
                  (second (first board)))
          t)
         (t (conflict n m (rest board)))))


;;this now prints a board no matter what order it is presented in
(defun print-board (board)
;  (sort board #'<= :key #'car)
  (format t "~%*")
  (print-horizontal-border board)
  (format t "*")
  (dotimes (row (length board))
    (format t "~%|")
    (dotimes (column (length board))
      (if (= column (second (assoc row board)))

          (format t " Q")
          (format t " .")))
    (format t " |"))
  (format t "~%*")
  (print-horizontal-border board)
  (format t "*"))


(defun print-horizontal-border (board)
  (dotimes (n (+ 1 (* 2 (length board))))
    (format t "-")))



;;;The following is Winston/Horn's original queen-finding algorithm.
;;;It does not discriminate sets of solutions that are closed under rotation
;;;and reflection.  Thus, a single solution can be equivalent to up to 8
other
;;;solutions.

(defun queen* (size &optional (board nil) (n 0) (m 0))
  (unless (= m size)
    ;;Check for conflict in current row and column
    (unless (conflict n m board)
      (if (= (+ 1 n) size)
        ;;If all queens placed, prin solution:
        (print-board (reverse (cons (list n m) board)))
        ;;Otherwise, proceed to next row:
        (queen* size (cons (list n m) board) (+ 1 n) 0)))
    ;;In any case, try with another column
    (queen* size board n (+ 1 m))))


;;;  This version improves on Winston/Horn in that it displays only one
instance
```

```
;;;of a closed set of solutions.  Thus, 8-queens solution is reduced from 96
to 12.
;;;It achieves this by using a bindings list of already-found solutions.
Since a change
;;;made at a deeper level of recursive search must be accessible to a higher
level,
;;;the bindings list was made a global parameter that is re-initialized when
queen is called.
;;;  This required an auxilliary function, thus labels.
;;;NO-REPEATS predicate returns t if no member of the set of solutions,
closed under reflection
;;;and rotation, is on the bindings list.
;;;Finally, the local variable new-board was assigned to save re-calculating
that data.


(defun queen (size)
  ;;initialize bindings variable
  (defparameter *bindings* nil)
  ;;declareauxilliary function
  (labels ((n-queen (size &optional (board nil) (n 0) (m 0))
             (unless (= m size)
               (let ((new-board (cons (list n m) board)))
                 (unless (conflict n m board)
                   (if (= (+ 1 n) size)
                     ;;check to see if solution is repeated
                     (if (no-repeats new-board (- size 1))
                       (progn                        ;only print solution if
not already printed
                         (print-board new-board)
                         (setf *bindings* (append (list (sort-board new-
board))
                                                  *bindings*))))
;change bindings
                     (n-queen size (cons (list n m) board) (+ 1 n) 0))))
                 (n-queen size board n (+ 1 m)))))
    ;;callauxilliary function:
          (n-queen size nil 0 0)))




(defun no-repeats (board size)
"looks to see if a permutation of current board has already been found--
assumes a properly sorted bindings list"


;(format t "~&Bindings are:   ~a" *bindings*)
;(format t "~&Equiv. Class is:  ~a" (make-set board size))

(if (equal *bindings* nil)     ;check to see if bindings in nil---find has a
problem with this
  t
  (labels ((find-double (-eq-class-)

             (if (equal nil -eq-class-)
               t
```

```
               (and (not (find (first -eq-class-) *bindings* :test #'equal))
;find solution in bindings
                    (find-double (rest -eq-class-))))))
    (find-double (make-set board size)))))
```

```
(defun reflect-horizontal (board size &optional (new-board nil))
"reflects across horizontal axis"

  (if (equal board nil)
    new-board
    (reflect-horizontal (rest board) size (cons  (list (- size (first (first
board)))
                                                       (second (first
board)))
                                                 new-board))))

(defun rotate-90 (board size &optional (new-board nil))
"reflects across diagonal axis"
  (if (equal board nil)
    (sort-board new-board)
    (rotate-90 (rest board) size (cons  (list (second (first board))
                                              (- size (first (first board))))
                                        new-board))))


(defun make-set (board size)
  (let*((board1 (sort-board board))
        (board2 (rotate-90 board1 size))
        (board3 (rotate-90 board2 size))
        (board4 (rotate-90 board3 size))
        (board5 (reflect-horizontal board1 size))
        (board6 (rotate-90 board5 size))
        (board7 (rotate-90 board6 size))
        (board8 (rotate-90 board7 size)))

    (list board1
          board2
          board3
          board4
          board5
          board6
          board7
          board8)))


(defun sort-board (board &optional (counter 0))
  "sorts a board configuration into ascending order by rows"
  (if (= counter (length board))
    nil
    (cons (assoc counter board)
          (sort-board board (+ counter 1)))))
```

5. Creating Hash Table in LISP

In Common LISP, hash table is a general-purpose collection. You can use arbitrary objects as a key or indexes.

When you store a value in a hash table, you make a key-value pair, and store it under that key. Later you can retrieve the value from the hash table using the same key. Each key maps to a single value, although you can store a new value in a key.

Hash tables, in LISP, could be categorized into three types, based on the way the keys could be compared - eq, eql or equal. If the hash table is hashed on LISP objects then the keys are compared with eq or eql. If the hash table hash on tree structure, then it would be compared using equal.

The **make-hash-table** function is used for creating a hash table. Syntax for this function is −

```
make-hash-table&key :test :size :rehash-size :rehash-threshold
```

Where −

- The **key** argument provides the key.
- The **:test** argument determines how keys are compared - it should have one of three values #'eq, #'eql, or #'equal, or one of the three symbols eq, eql, or equal. If not specified, eql is assumed.
- The **:size** argument sets the initial size of the hash table. This should be an integer greater than zero.
- The **:rehash-size** argument specifies how much to increase the size of the hash table when it becomes full. This can be an integer greater than zero, which is the number of entries to add, or it can be a floating-point number greater than 1, which is the ratio of the new size to the old size. The default value for this argument is implementation-dependent.
- The **:rehash-threshold** argument specifies how full the hash table can get before it must grow. This can be an integer greater than zero and less than the :rehash-size (in which case it will be scaled whenever the table is grown), or it can be a floating-point number between zero and 1. The default value for this argument is implementation-dependent.

6. Reading Input from Keyboard

The **read** function is used for taking input from the keyboard. It may not take any argument.

For example, consider the code snippet −

```
(write ( + 15.0 (read)))
```

Assume the user enters 10.2 from the STDIN Input, it returns,

```
25.2
```

The read function reads characters from an input stream and interprets them by parsing as representations of Lisp objects.

Example

Create a new source code file named main.lisp and type the following code in it −

```
;thefunctionAreaOfCircle
; calculates area of a circle
;when the radius is input from keyboard

(defunAreaOfCircle()
(terpri)
(princ"Enter Radius: ")
(setq radius (read))
(setq area (*3.1416 radius radius))
(princ"Area: ")
(write area))
(AreaOfCircle)
```

When you execute the code, it returns the following result −

```
Enter Radius: 5 (STDIN Input)
Area: 78.53999
```

7. Reading Input from Keyboard

The **read** function is used for taking input from the keyboard. It may not take any argument.

For example, consider the code snippet −

```
(write ( + 15.0 (read)))
```

Assume the user enters 10.2 from the STDIN Input, it returns,

```
25.2
The read function reads characters from an input stream and interprets them
by parsing as representations of Lisp objects.
```

Example

Create a new source code file named main.lisp and type the following code in it −

```
;thefunctionAreaOfCircle
; calculates area of a circle
;when the radius is input from keyboard

(defunAreaOfCircle()
(terpri)
(princ"Enter Radius: ")
(setq radius (read))
(setq area (*3.1416 radius radius))
(princ"Area: ")
```

```
(write area))
(AreaOfCircle)
```

When you execute the code, it returns the following result −

```
Enter Radius: 5 (STDIN Input)
Area: 78.53999
```

Exercise

1. Explain how binary trees are implemented.
2. Explain how elements are search using tree.
3. Explain how hash table is implemented.
4. Write a function for Tower of Hanoi.
5. Explain 8-queens problem using Lisp.

Chapter 7
Fuzzy Systems

7.1 Introduction

Fuzzy logic provides an inference morphology that enables approximate human reasoning capabilities to be applied to knowledge-based systems. The theory of fuzzy logic provides a mathematical strength to capture the uncertainties associated with human cognitive processes, such as thinking and reasoning. Fuzzy sets were introduced by Zadeh (1965).

Characteristics of fuzzy logic:

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

7.1.1 Fuzzy Systems: History

- In 300 years BC, Aristotle came up with binary logic involving the numbers 0 and 1.
- It came down to one law: True or False.
- Later Plato questioned the rationale of it by proposing a third region which is beyond True or false.
- Buddha supported this argument by stating that world as it is, filled with contradictions, with things and not things.
- Philosophers like Hegel, Marx and Engels supported the school of thought of many valued logic.
- In 1900, Lukasiewicz first proposed three value logic along with the mathematics to accompany it.
- The third value he proposed was “possible” and he assigned it a numerical value between True and False.
- In 1965 Dr. Lotfi A Zadeh published his work “Fuzzy Sets”, which described the mathematics of fuzzy set theory.
- This theory proposed the membership function operate over the range of real numbers [0, 1], in place of 0 and 1 as followed by Boolean Set theory.

- The indicator function of a non fuzzy set A is given by
- I_A = 1 if x ε A
- = 0 if x ε A.
- Zadeh extended this function to multi value membership function m_A: x → [0, 1].
- This membership function measures the degree to which element x belongs to the set A.
- $m_A(x)$ = Degree (x ε A)
- $m_A(x)$ = 0 denotes that x is not a member of the set;
- $m_A(x)$ = 1 denotes that x is definitely a member;
- And all other values denote degrees of membership.

Fuzzy systems are more favorable in that their behavior can be explained based on fuzzy rules and thus their performance can be adjusted by tuning the rules. But since, in general, knowledge acquisition is difficult and also the universe of discourse of each input variable needs to be divided into several intervals, applications of fuzzy systems are restricted to the fields where an expert knowledge is available and the number of input variables is small.

What is Fuzzy Logic?

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

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

In the given figure 7.1 what do we recognize?

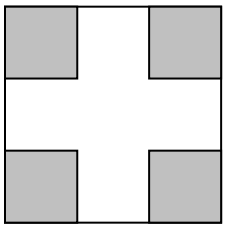


Figure 7.1: Illusion

We 'recognize' an illusionary bright cross. But technically, there is no cross. Only four squares within a square. Such a square exists only in our brain, not on the

screen. The real time interaction of millions of neurons in our brain is behind this cross like perception. The asynchronous, nonlinear, massively parallel, distributed neurons perform such recognition under uncertainty. We reason something with vague concepts, beliefs, estimates, guesses etc. This inexactness is called fuzzy. Though we use exact scientific tools in day to day decision making, the final control remains fuzzy. E.g. medical diagnosis.
This we may casually refer as experience, judgment, sixth sense etc.

In other words, we can say that fuzzy logic is not logic that is fuzzy, but logic that is used to describe fuzziness.

7.2 Set Theory

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

Example

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

7.2.1 Mathematical Representation of a Set

Sets can be represented in two ways −

1. **Roster or Tabular Form**

In this form, a set is represented by listing all the elements comprising it. The elements are enclosed within braces and separated by commas.

Following are the examples of set in Roster or Tabular Form −

- Set of vowels in English alphabet, A = {a,e,i,o,u}
- Set of odd numbers less than 10, B = {1,3,5,7,9}

2. **Set Builder Notation**

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

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

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

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

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

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

Example − If S = {1,1.2,1.7,2},1 ∈ S but 1.5 ∉ S

3. Cardinality of a Set

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

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

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

$|X| \leq |Y|$ denotes that set X's cardinality is less than or equal to set Y's cardinality. It occurs when the number of elements in X is less than or equal to that of Y. Here, there exists an injective function 'f' from X to Y.

$|X| < |Y|$ denotes that set X's cardinality is less than set Y's cardinality. It occurs when the number of elements in X is less than that of Y. Here, the function 'f' from X to Y is injective function but not bijective.

If $|X| \leq |Y|$ and $|X| \leq |Y|$ then $|X| = |Y|$. The sets X and Y are commonly referred as **equivalent sets**.

7.3 Types of Sets

Sets can be classified into many types; some of which are finite, infinite, subset, universal, proper, singleton set, etc.

a. Finite Set

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

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

b. Infinite Set

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

Example − S = {x|x ∈ N and x > 10}

c. Subset

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

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

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

d. Proper Subset

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

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

e. Universal Set

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

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

f. Empty Set or Null Set

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

Example – S = {x|x ∈ N and 7 < x < 8} = Φ

g. Singleton Set or Unit Set

A Singleton set or Unit set contains only one element. A singleton set is denoted by {s}.

Example − S = {x|x ∈ N, 7 < x < 9} = {8}

h. Equal Set

If two sets contain the same elements, they are said to be equal.

Example − If A = {1,2,6} and B = {6,1,2}, they are equal as every element of set A is an element of set B and every element of set B is an element of set A.

i. Equivalent Set

If the cardinalities of two sets are same, they are called equivalent sets.

Example − If A = {1,2,6} and B = {16,17,22}, they are equivalent as cardinality of A is equal to the cardinality of B. i.e. |A| = |B| = 3

j. Overlapping Set

Two sets that have at least one common element are called overlapping sets. In case of overlapping sets −

$$n(A \cup B) = n(A) + n(B) - n(A \cap B)$$
$$n(A \cup B) = n(A - B) + n(B - A) + n(A \cap B)$$
$$n(A) = n(A - B) + n(A \cap B)$$
$$n(B) = n(B - A) + n(A \cap B)$$

Example − Let, A = {1,2,6} and B = {6,12,42}. There is a common element '6', hence these sets are overlapping sets.

k. Disjoint Set

Two sets A and B are called disjoint sets if they do not have even one element in common. Therefore, disjoint sets have the following properties −

$$n(A \cap B) = \phi$$
$$n(A \cup B) = n(A) + n(B)$$

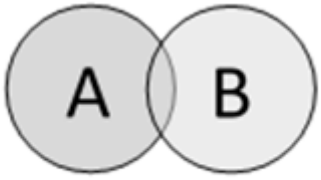
7.4 Operations on Classical Sets

Set Operations include Set Union, Set Intersection, Set Difference, Complement of Set, and Cartesian Product.

1. Union

The union of sets A and B (denoted by A ∪ BA ∪ B) is the set of elements which are in A, in B, or in both A and B. Hence, A ∪ B = {x|x ∈ A OR x ∈ B}.

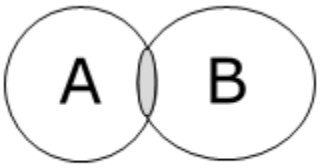
Example − If A = {10,11,12,13} and B = {13,14,15}, then A ∪ B = {10,11,12,13,14,15} – The common element occurs only once.



A ∪ B

2. Intersection

The intersection of sets A and B (denoted by A ∩ B) is the set of elements which are in both A and B. Hence, A ∩ B = {x|x ∈ A AND x ∈ B}.

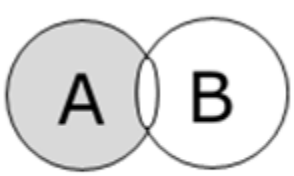


A ∩ B

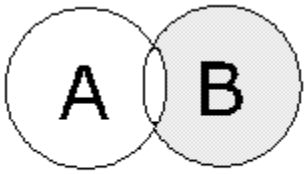
3. Difference/ Relative Complement

The set difference of sets A and B (denoted by A–B) is the set of elements which are only in A but not in B. Hence, A − B = {x|x ∈ A AND x ∉ B}.

Example − If A = {10,11,12,13} and B = {13,14,15}, then (A − B) = {10,11,12} and (B − A) = {14,15}. Here, we can see (A − B) ≠ (B − A)



A - B



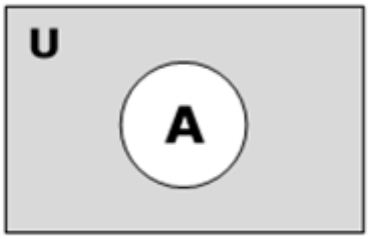
B - A

4. Complement of a Set

The complement of a set A (denoted by A′) is the set of elements which are not in set A. Hence, A′ = {x|x ∉ A}.

More specifically, A′ = (U−A) where U is a universal set which contains all objects.

Example − If A = {x|x belongs to set of add integers} then A′ = {y|y does not belong to set of odd integers}



Complement of set A

5. Cartesian Product / Cross Product

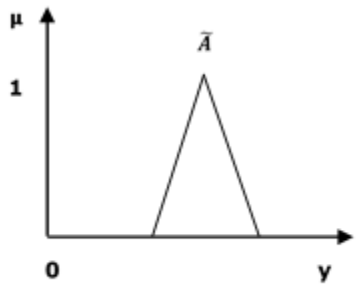
The Cartesian product of n number of sets A1,A2,…An denoted as A1 × A2...× An can be defined as all possible ordered pairs (x1,x2,…xn) where x1 ∈ A1,x2 ∈ A2,…xn ∈ An

Example − If we take two sets A = {a,b} and B = {1,2},

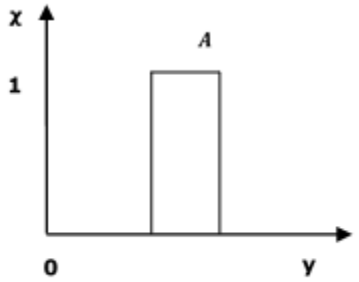
The Cartesian product of A and B is written as − A × B = {(a,1),(a,2),(b,1),(b,2)}

And, the Cartesian product of B and A is written as − B × A = {(1,a),(1,b),(2,a),(2,b)}

Fuzzy sets can be considered as an extension and gross oversimplification of classical sets. It can be best understood in the context of set membership. Basically it allows partial membership which means that it contain elements that have varying degrees of membership in the set. From this, we can understand the difference between classical set and fuzzy set. Classical set contains elements that satisfy precise properties of membership while fuzzy set contains elements that satisfy imprecise properties of membership.



Membership Function of Fuzzy set $\tilde{A}$



Membership Function of classical set A

7.5 Properties of Fuzzy Sets

Let us discuss the different properties of fuzzy sets.

1. Commutative Property

Having two fuzzy sets A˜ and B˜, this property states − A˜∪B˜ = B˜∪A˜

A˜∩B˜=B˜∩A˜

2.Associative Property

Having three fuzzy sets A˜, B˜ and C˜ , this property states −

A˜∪(B˜∪C˜)=(A˜∪B˜)∪C˜
A˜∩(B˜∩C˜)=(A˜∪B˜)∪C˜

3.Distributive Property

Having three fuzzy sets $\tilde{A}$, $\tilde{B}$ and $\tilde{C}$, this property states −

$$\tilde{A}\cup(\tilde{B}\cap\tilde{C})=(\tilde{A}\cup\tilde{B})\cap(\tilde{A}\cup\tilde{C})$$
$$\tilde{A}\cap(\tilde{B}\cup\tilde{C})=(\tilde{A}\cap\tilde{B})\cup(\tilde{A}\cap\tilde{C})$$

4.Idempotency Property

For any fuzzy set $\tilde{A}$, this property states −

$$\tilde{A}\cup\tilde{A}=\tilde{A}$$
$$\tilde{A}\cap\tilde{A}=\tilde{A}$$

5.Identity Property

For fuzzy set $\tilde{A}$ and universal set U , this property states −

$$\tilde{A}\cup\varphi=\tilde{A}$$
$$\tilde{A}\cap U=\tilde{A}$$
$$\tilde{A}\cap\varphi=\varphi$$
$$\tilde{A}\cup U=U$$

6.Transitive Property

Having three fuzzy sets $\tilde{A}$, $\tilde{B}$ and $\tilde{C}$, this property states −

$$\text{If } \tilde{A}\subseteq\tilde{B}\subseteq\tilde{C}, \text{ then } \tilde{A}\subseteq\tilde{C}$$

7.Involution Property

For any fuzzy set $\tilde{A}$, this property states −

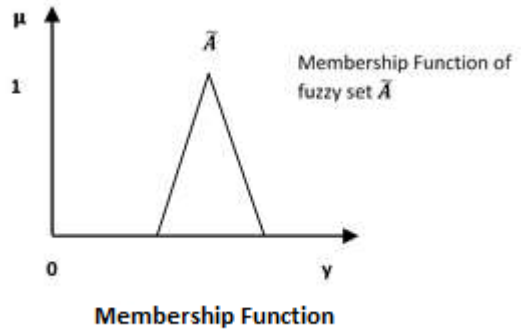
$$\overline{\overline{\tilde{A}}}=\tilde{A}$$

8.De Morgan's Law

This law plays a crucial role in proving tautologies and contradiction. This law states −

$$\overline{\tilde{A}\cap\tilde{B}}=\overline{\tilde{A}}\cup\overline{\tilde{B}}$$
$$\overline{\tilde{A}\cup\tilde{B}}=\overline{\tilde{A}}\cap\overline{\tilde{B}}$$

We already know that fuzzy logic is not logic that is fuzzy but logic that is used to describe fuzziness. This fuzziness is best characterized by its membership function. In other words, we can say that membership function represents the degree of truth in fuzzy logic.



Membership Function

Following are a few important points relating to the membership function −

- Membership functions were first introduced in 1965 by Lofti A. Zadeh in his first research paper "fuzzy sets".
- Membership functions characterize fuzziness (i.e., all the information in fuzzy set), whether the elements in fuzzy sets are discrete or continuous.
- Membership functions can be defined as a technique to solve practical problems by experience rather than knowledge.
- Membership functions are represented by graphical forms.
- Rules for defining fuzziness are fuzzy too.

6.6 Mathematical Notation

We have already studied that a fuzzy set $\tilde{A}$ in the universe of information U can be defined as a set of ordered pairs and it can be represented mathematically as −

$$\tilde{A}=\{(y,\mu_{\tilde{A}}(y))|y\in U\}$$

Here $\mu\tilde{A}(\cdot)$ = membership function of $\tilde{A}$; this assumes values in the range from 0 to 1, i.e., $\mu\tilde{A}(\cdot)\in[0,1]$. The membership function $\mu\tilde{A}(\cdot)$ maps U to the membership space M.

The dot $(\cdot)$ in the membership function described above, represents the element in a fuzzy set; whether it is discrete or continuous.

6.7 Features of Membership Functions

We will now discuss the different features of Membership Functions.

6.7.1 Core

For any fuzzy set $\tilde{A}$, the core of a membership function is that region of universe that is characterize by full membership in the set. Hence, core consists of all those elements y

of the universe of information such that,

$$\mu_{\tilde{A}}(y)=1$$

6.7.2 Support

For any fuzzy set $\tilde{A}$

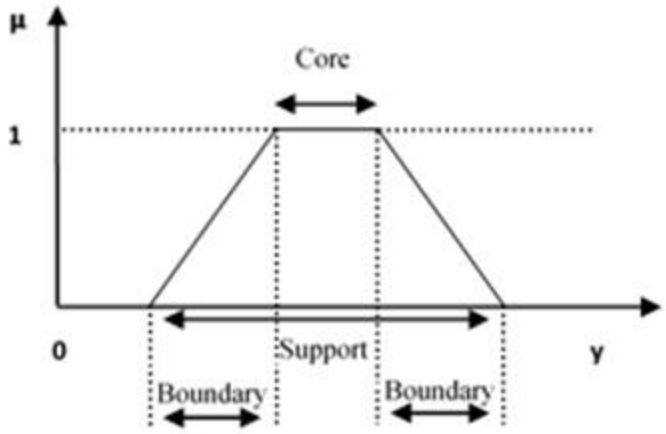
, the support of a membership function is the region of universe that is characterize by a nonzero membership in the set. Hence core consists of all those elements y of the universe of information such that,

$$\mu_{\tilde{A}}(y)>0$$

6.7.3 Boundary

For any fuzzy set $\tilde{A}$, the boundary of a membership function is the region of universe that is characterized by a nonzero but incomplete membership in the set. Hence, core consists of all those elements y of the universe of information such that,

$$1>\mu_{\tilde{A}}(y)>0$$



Features of Membership Function

6.8 Fuzzification

It may be defined as the process of transforming a crisp set to a fuzzy set or a fuzzy set to fuzzier set. Basically, this operation translates accurate crisp input values into linguistic variables.

Following are the two important methods of fuzzification −

6.8.1 Support Fuzzification (s-fuzzification) Method

In this method, the fuzzified set can be expressed with the help of the following relation −

$$\tilde{A}=\mu_1 Q(x_1)+\mu_2 Q(x_2)+...+\mu_n Q(x_n)$$

Here the fuzzy set $Q(x_i)$

is called as kernel of fuzzification. This method is implemented by keeping μ_i constant and x_i being transformed to a fuzzy set $Q(x_i)$

.

6.8.2 Grade Fuzzification (g-fuzzification) Method

It is quite similar to the above method but the main difference is that it kept x_i

constant and μ_i is expressed as a fuzzy set.

6.9 Defuzzification

It may be defined as the process of reducing a fuzzy set into a crisp set or to convert a fuzzy member into a crisp member. We have already studied that the fuzzification process involves conversion from crisp quantities to fuzzy quantities. In a number of engineering applications, it is necessary to defuzzify the result or rather "fuzzy result" so that it must be converted to crisp result. Mathematically, the process of Defuzzification is also called "rounding it off".

The different methods of Defuzzification are described below −

6.9.1 Max-Membership Method

This method is limited to peak output functions and also known as height method. Mathematically it can be represented as follows −

$$\mu_{\tilde{A}}(x_*) > \mu_{\tilde{A}}(x) for all x \in X$$

Here, x_* is the defuzzified output.

6.9.2 Centroid Method

This method is also known as the center of area or the center of gravity method. Mathematically, the defuzzified output x_*

will be represented as −

$$x_* = \int \mu_{\tilde{A}}(x).x dx \int \mu_{\tilde{A}}(x).dx$$

6.9.3 Weighted Average Method

In this method, each membership function is weighted by its maximum membership value. Mathematically, the defuzzified output x_*

will be represented as −

$$x_* = \sum \bar{\mu}_{\tilde{A}}(x_i).x_i \sum \overline{\mu}_{\tilde{A}}(x_i)$$

6.9.4 Mean-Max Membership

This method is also known as the middle of the maxima. Mathematically, the defuzzified output $x*$

will be represented as −

$$x*=\sum_{i=1}n\overline{x_i}\ n$$

6.10 Propositions in Fuzzy Logic

As we know that propositions are sentences expressed in any language which are generally expressed in the following canonical form −

s as P

Here, s is the Subject and P is Predicate. For example, “Delhi is the capital of India”, this is a proposition where “Delhi” is the subject and “is the capital of India” is the predicate which shows the property of subject.

We know that logic is the basis of reasoning and fuzzy logic extends the capability of reasoning by using fuzzy predicates, fuzzy-predicate modifiers, fuzzy quantifiers and fuzzy qualifiers in fuzzy propositions which creates the difference from classical logic.

Propositions in fuzzy logic include the following −

6.11 Fuzzy Events, Fuzzy Means and Fuzzy Variances

With the help of an example, we can understand the above concepts. Let us assume that we are a shareholder of a company named ABC. And at present the company is selling each of its share for ₹40. There are three different companies whose business is similar to ABC but these are offering their shares at different rates - ₹100 a share, ₹85 a share and ₹60 a share respectively.

Now the probability distribution of this price takeover is as follows −

Price	₹100	₹85	₹60
Probability	0.3	0.5	0.2

Now, from the standard probability theory, the above distribution gives a mean of expected price as below –

$$100\times0.3+85\times0.5+60\times0.2=84.5$$

And, from the standard probability theory, the above distribution gives a variance of expected price as below −

$$(100-84.5)2\times0.3+(85-84.5)2\times0.5+(60-84.5)2\times0.2=124.825$$

Suppose the degree of membership of 100 in this set is 0.7, that of 85 is 1, and the degree of membership is 0.5 for the value 60. These can be reflected in the following fuzzy set −

$$\{0.7100,185,0.560,\}$$

The fuzzy set obtained in this manner is called a fuzzy event.

We want the probability of the fuzzy event for which our calculation gives −

$$0.7\times0.3+1\times0.5+0.5\times0.2=0.21+0.5+0.1=0.81$$

Now, we need to calculate the fuzzy mean and the fuzzy variance, the calculation is as follows −

Fuzzy_mean $=(10.81)\times(100\times0.7\times0.3+85\times1\times0.5+60\times0.5\times0.2) =85.8$

Fuzzy_Variance $=7496.91-7361.91=135.27$

FUZZY SYSTEMS: EXAMPLE

Let us consider the example of TALL to illustrate fuzzy set.

We can assign a degree of membership to each person in the fuzzy set TALL as follows:

Tall(x) = 0, if height(x) < 5’
= (height(x) - 5’)/2, if 5’ <= height(x) <= 7’
= 1, if height(x) > 7’.

The heights and degrees of membership of each person can be shown as follows:

Person	Height	Degree of Membership
A	5’ 2”	0.10
B	5’ 3”	0.15

C	5’ 5”	0.25
D	5’ 7”	0.35
E	6’ 1”	0.54

7.12 Some Applications / Products

- Railway subway in Sendai, Japan where train’s movements are controlled by fuzzy controlled systems.
- Omron – Camera aiming for the telecast of sporting events.
- Hitachi – washing machine with single button control.
- Sony – pocket computers with handwritten recognition.

Exercise:

1. What is Fuzzy System? State its Applications.
2. Explain fuzzy set theory with suitable example.
3. Explain types of fuzzy set.
4. Explain properties of fuzzy set.
5. Explain in brief about membership function.
6. Write a note on Defuzzification.
7. Explain in brief about fuzzification.

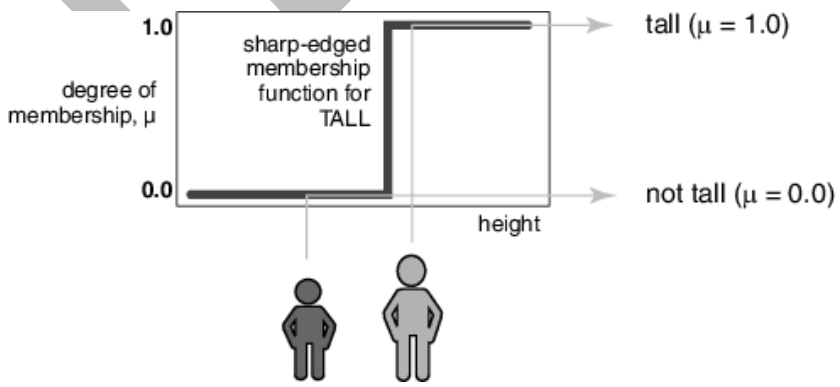
Chapter 8
Fuzzy Logic Concept and System

8.1 The Fuzzy Logic Concept

The way we perceive the world is continually changing and cannot always be defined in true or false statements. Take for example the set of all the mangoes and all the mangoes cores in the world. Now take one of those mangoes; it belongs to the set of all mangoes. Now take a bite out of that mango; it is still mango right? If so, it still belongs to the set of mangoes. After several more bites have been taken and you are left with a mango core and it belongs to the set of mango cores. At what point did the mango cross over from being mango to being mango core? What if you could get one more bite out of that mango core, does that move it into a different set?

The definition of the mango and mango core sets are too strictly defined when looking at the process of eating amango. The area between the two sets is not clearly defined since the object cannot belong to the set of mangoes and mango cores because, by definition, a mango core is NOT amango. The sets defining mangoes and mango cores need to be redefined as fuzzy sets.

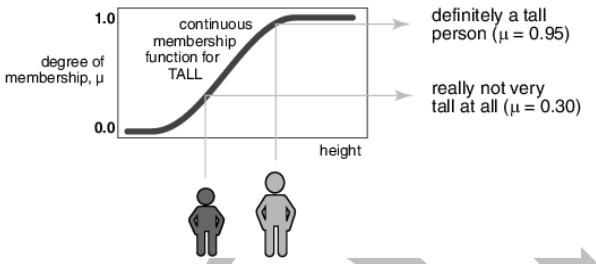
Fuzziness is explored as an alternative to randomness for describing uncertainty. In mathematics a set, by definition, is a collection of things that belong to some definition. Any item either belongs to that set or does not belong to that set. Let us look at another example; the set of tall men. We shall say that people taller than or equal to 6 feet are tall. This set can be represented graphically as follows:



The function shown above describes the membership of the 'tall' set, you are either in it or you are not in it. This sharp edged membership functions works nicely for binary operations and mathematics, but it does not work as nicely in describing the real world. The membership function makes no distinction between

somebody who is 6'1" and someone who is 7'1", they are both simply tall. Clearly there is a significant difference between the two heights. The other side of this lack of distinction is the difference between a 5'11" and 6' man. This is only a difference of one inch, however this membership function just says one is tall and the other is not tall.

The fuzzy set approach to the set of tall men provides a much better representation of the tallness of a person. The set, shown below, is defined by a continuously inclining function.



The membership function defines the fuzzy set for the possible values underneath of it on the horizontal axis. The vertical axis, on a scale of 0 to 1, provides the membership value of the height in the fuzzy set. So for the two people shown above the first person has a membership of 0.3 and so is not very tall. The second person has a membership of 0.95 and so he is definitely tall. He does not, however, belong to the set of tall men in the way that bivalent sets work; he has a high degree of membership in the fuzzy set of tall men.

8.1.1 Bivalent Logic Creates Paradoxes

In formal logic, the principle of bivalence becomes a property that a semantics may or may not possess. It is not the same as the law of excluded middle, however, and a semantics may satisfy that law without being bivalent.The intended semantics of classical logic is bivalent, but this is not true of every semantics for classical logic. In Boolean-valued semantics (for classical propositional logic), the truth values are the elements of an arbitrary Boolean algebra, "true" corresponds to the maximal element of the algebra, and "false" corresponds to the minimal element.

- A man says “Don’t trust me”. Can we trust him?
- One side of a card says “The sentence on the other side is true” and the other side of the card says “The sentence on the other side is not true”. Which side is true?
- A speaker tells “I lie”. Does he tell the truth?
- A liar says all his friends are liars. Does he lie?
- A barber shaves everybody who cannot shave themselves. Can he shave himself?

8.1.2 Bivalent Paradoxes as Fuzzy Mid Points

Consider the bivalent paradoxes again. A bumper sticker on the truck reads TRUST ME. Suppose instead a sticker reads DON'T TRUST ME. Should we trust the driver? If we do, then, as the bumper sticker instructs, we do not. But if we don't trust the driver, then, again in accord with the bumper sticker, we do trust the driver. The classical liar paradox has the same form. Does the liar X lie when he says that all Men's are liars? If he lies, he tells the truth. If he tells the truth, he lies. A barber is inPune advertises his services with the logo "I shave all, and only, those men who don't shave themselves." Who shaves the barber? If he shaves himself, then according to his logo he does not. If he does not, then according to his logo he does. Consider the card that says on one side "The sentence on the other side is true," and says on the other side, "The sentence on the other side is false."

The "paradoxes" have the same form. A statement s and its negation not-s have the same truth-value t(s):

$t(s) = t(not\text{-}s)$…….. (1)

The two statements are both TRUE (1) and both FALSE (0). This violates the laws of non-contradiction and excluded middle. For bivalent truth tables it is reminded that negation reverses truth-value:

$t(not\text{-}s) = 1 - t(s)$……..(2)

So (1) reduces to $t(s) = 1 - t(s)$……(3)

If S is true, t(S)=1 and t(not S)=0, then 1=0.

If t(S)=0 it also implies the contradiction 1=0.

The fuzzy or multivalued interpretation accepts the logical relation (1-3) and,instead of insisting that t(S)=0 or t(S)=1, simply solves for t(S) in (1-3):

$2t(S) = 1$ (1-4)

or

$t(S) = 1/2$ (1-5)

So the "paradoxes" reduce to literal half-truths. They represent in the extreme the uncertainty inherent in every empirical statement. Geometrically, the fuzzy approach places the paradoxes at the midpoint of the one-dimensional unit

hypercube [0,1]. More general paradoxes reside at the midpoint of n-dimensional hypercube, the unique point equidistant to all 2^n vertices.

Multivaluedness also resolves the classical sorties paradoxes. Consider a heap of sand. Is it still a heap if we remove one grain of sand? How about two grains? Three? If we argue bivalent by induction, we eventually remove all grains and still conclude that a heap remains, or that it has suddenly vanished. No single grain takes us from a heap to no heap. The same holds if we pluck out hairs from a non-bald scalp or remove 5%, 10%, or more of the molecules from a table or brain. We transition gradually, not abruptly, from a thing to its opposite. Physically we experience degrees of occurrence. Suppose there are n grains of sand in the heap. Removing one grain leaves n-1 grains and a truth-value $t(S_n - 1)$ of the statement S n – 1 and implies n-1 sand grains are heap. In general the truth-value $t (S_n - 1)$ obeys $t (S_n-1) <1$. $t(S_n-1)$ may be close to unity, but we have some nonzero doubt d_{n-1} about the truth of the matter.
For instance $t(S_n) = 1 - d_n(1-6)$, Where $0<=d_n<=d_{n-1} <=…. <=d_{n-m} <=…<=1$.

So $t(S_{n-m})$ approaches zero as m increases to n. If we argue inductively, we can interpret the overall inference as the forward chain "(If S_n, then S_{n-1} and (If S_{n-1},then S_{n-2} and…. and(If S_1, then S_0)".
Fuzzy Set Theory defines Fuzzy Operators on Fuzzy Sets. The problem in applying this is that the appropriate Fuzzy Operator may not be known. For this reason, Fuzzy logic usually uses IF/THEN rules, or constructs that are equivalent, such as fuzzy associative matrices.

Rules are usually expressed in the form:
IF *variable* IS *set* THEN *action*

For example, an extremely simple temperature regulator that uses a fan might look like this:
IF temperature IS very cold THEN stop fan
IF temperature IS cold THEN turn down fan
IF temperature IS normal THEN maintain level
IF temperature IS hot THEN speed up fan

Notice there is no "ELSE". All of the rules are evaluated, because the temperature might be "cold" and "normal" at the same time to differing degrees.

The AND, OR, and NOT operators of boolean logic exist in fuzzy logic, usually defined as the minimum, maximum, and complement; when they are defined this way, they are called the *Zadeh operators*, because they were first defined as such in Zadeh's original papers. So for the fuzzy variables x and y:

NOT x = (1 - truth(x))

x AND y = minimum(truth(x), truth(y))

x OR y = maximum(truth(x), truth(y))

8.2 Fuzziness in the twentieth century

Logical paradoxes and the Heisenberg uncertainty principle led to thedevelopment of multivalued or "fuzzy" logic in the 1920s and 1930s. Quantumtheorists allowed for indeterminacy by including a third or middle truth-value in the bivalent logical framework. The next step allowed degrees of indeterminacy, viewing TRUE and FALSE as the two limiting cases of the spectrum of indeterminacy.

Polish logician Jan Lukasiewicz first normally developed a three-valued logical system in the early 1930s. He extended the range of truth-values from {0,1,1/2} to all rational numbers in {0,1}, and finally to all numbers in {0,1} itself. Logics that use the general truth function t:{Statements} [0,1] define continuous or "fuzzy" logics.

In the 1930s quantum philosopher max black applied continuous logic componentwise to sets or lists of elements or symbols. Historically, black drew the firstfuzzy-set membership functions. Black called the uncertainty of these structuresvagueness. Anticipating Zadeh's fuzzy –set theory, each element in black'smultivalued sets and lists behaved as a statement in a continuous logic.

Zadeh extended the bivalent indicator function I_a of non-fuzzy subset a of x, $I_A(x)$={1 if x∈A | 0 if x∉A
To a multivalued indicator or membership function m_a : X €[0.1]. This allows us to combine such multivalued or fuzzy sets with the pointwise operators of indicator functions :
I_AB (x) = min (I_A(x) , I_B(x))I_A∈B (x) = max (I_A(x) , I_B(x))IA_c (x) = 1 - I_A (x) A|Biff I_A (x) I_B (x) for all x in X The membership value m A (x) measures the elementhood or degree to which element x belongs to set A: mA(x) = Degree (x | A) Just as the individual indicator values I A (x) behave as statements in bivalent prepositional calculus, membership values m A (x) correspond to statements in continuous logic.
Sets as Points in Cubes: Fuzziness prevents logical certainty at the level at the level of black-white axioms. This seems unsettling to some and liberating to others.
Neural networks and fuzzy systems process inexact information and process it inexactly. Neural networks recognize ill-defined patterns without an explicit set of

rules.

Fuzzy systems estimate functions and control systems with partial descriptions of system behavior. The neuronal state space, the set of all possible neural outputs, equals the set of all n-dimensional fit vectors, the fuzzy power set. Both equal the unit hypercube I n =[0,1] n =[0,1] X…X [0,1], the set of all vectors of length n and with coordinates inthe unit interval [0,1]. The 2n vertices of In represent extreme neuronal-output combinations. Themidpoint of the cube, where a fuzzy set A equals its own opposite Ac, hasmaximum fuzzy entropy. The black-white vertices have minimal fuzzy entropy. Proper fuzzy sets, non-vertex points, A violate the law of non-contradiction and excludedmiddle:$A \in$ Acf and A È Ac ^{1} X

Fuzzy power set F (2x) of X corresponds to the unit square when X={x1, x2}. The four non-fuzzy subsets in the nonfuzzy power set 2X correspond to the four corners of the 2-cube. The fuzzy subset A correspond to the fit vector (1/3, 3/4) and to a point inside the 2-cube if mA (x1)=1/3 and mA (x2)=3/4. The midpoint M of the unit square corresponds to the maximally fuzzy set. Long diagonals connect nonfuzzy set complements. Fuzziness in a probabilistic world

Is uncertainty same as the randomness? If we are not sure about something, is it only up to chance? Do the notions of likelihood and probability exhaust our notions of uncertainity? Many people trained in probability and statistics believe so. Some even say so, and say so loudly.Randomness and fuzziness differ conceptually and theoretically. We canillustratesome differences with examples.Randomness and fuzziness also share many similarities. Both systems describe uncertainty with numbers in the unit interval [0,1]. This ultimately means that both systems describe uncertainty numerically. Both systems combine sets and propositions associatively, commutatively, and distributively. The key distinction concerns how the systems jointly treat a set A and its opposite Ac. Classical set theory demands AÇ Ac = f, and probability theory conforms: p (A Ç Ac)=p (f)=0. So A Ç Ac represents aprobabilistic impossible event. But fuzziness begins when AÇAc¹.f

RandomnessVs.Ambiguity: Whether VS. How Much

Fuzziness describes event ambiguity. It measures the degree to which an event occurs, not whether it occurs. Randomness describes the uncertainty of event occurrence. An event occurs or not, and you can bet on it. The issue concerns the occurring event: Is it uncertain in any way? Can we unambiguously distinguish theevent from its opposite?Whether an event occurs is “random”. To what degree it occurs is fuzzy. Whether an ambiguous event occurs – as when we say there is 20% chance of light rain tomorrow-involves compound uncertainties, the probability of a fuzzy event. We regularly apply probabilities to fuzzy events: small errors, satisfied customers, A students and galactic clusters etc. we understand that, at least around the edges, some satisfied customers can be somewhat unsatisfied, some A students might equally be B+ students, some stars are as much in a galactic cluster as out of it. Events can transition more or less smoothly to their opposites, making classification hard near the midpoint of the transition. But in theory-in formal descriptions and in textbooks-the events and

their opposites are black and white. A hill is a mountain if it is at least x meters tall, not a mountain if it is one micron lessthan x in height.Consider some further examples. The probability that this chapter gets published is one thing. The degree to which it gets published is another. Suppose there is 50% chance that there is an apple in the refrigerator (electron in a cell). That is one state of affairs, perhaps arrived at through frequency calculations. Now suppose there is half an apple in the refrigerator. That is another state of affairs. Both state of affairs are superficially equivalent in terms of their numerical uncertainty. Yet physically, ontologically, they differ. One is "random" the other fuzzy. Consider parking your car in a parking lot with painted parking spaces. You can park in any space with some probability. Your car will totally occupy one space and totally unoccupy all other spaces. The probability number reflects a frequency history or Bayesian brain state that summarizes which parking space your car will totally occupy. Alternatively, you can park in every space to some degree. Your car will partially and deterministically, occupy every space. In practice your car will occupy most spaces to zero degree. Finally, we can use numbers in [0,1] to describe, for each parking space, the occurrence probability of each degree of partialoccupancy- probabilities of fuzzy events.If we assume events as unambiguous, as in balls-in-urn experiments, there is no set fuzziness. Only randomness remains. But when we discuss the physical universe, every assertion of event ambiguity or unambiguity is an empirically hypothesis. We habitually overlook this when we apply probability theory. Years of such oversight have entrenched the sentiment that uncertainty is randomness, and randomness alone. We systematically assume away event ambiguity. We call the partially empty glass empty and call the small number zero. This silent assumption of universal non-ambiguity resembles the pre-relativistic assumption of an uncurved universe. Consider the inexact oval in figure below. Does it make more sense to say that the oval is probably an ellipse, or that it is a fuzzy ellipse? There seems nothing random about the matter. The situation is deterministic: All the facts are in. yet uncertainty remains. The uncertainty arises from the simultaneous occurrence of two properties: to some extent the inexact oval is ellipse, and to some extent it is not an ellipse.

8.3 Fuzziness systems and applications

Fuzzy systems store banks of fuzzy associations or common-sense "rules". A fuzzy traffic controller might contain the fuzzy association. "If traffic is heavy in this direction, then keep the light green longer ". Fuzzy phenomena admit degrees. Some traffic configurations are heavier than others. Some green-light durations are longer than others. The single fuzzy association (HEAVY, LONGER) encodes all these combinations. Fuzzy systems are even newer than neural systems. Fuzzy systems "intelligently" automate subways; focus cameras and camcorders; tune color televisions and computer disc heads; control automobile transmissions, cruise controllers, and emergency braking systems; defrost refrigerators; control air conditioners; automate washing machines and vacuum sweepers; guide robot-arm manipulators; invest in securities; control

traffic lights, elevators, and cement mixers; recognize kanji characters; select golf clubs; even arrange flowers.

8.4 Fuzzy Systems & Neural Networks

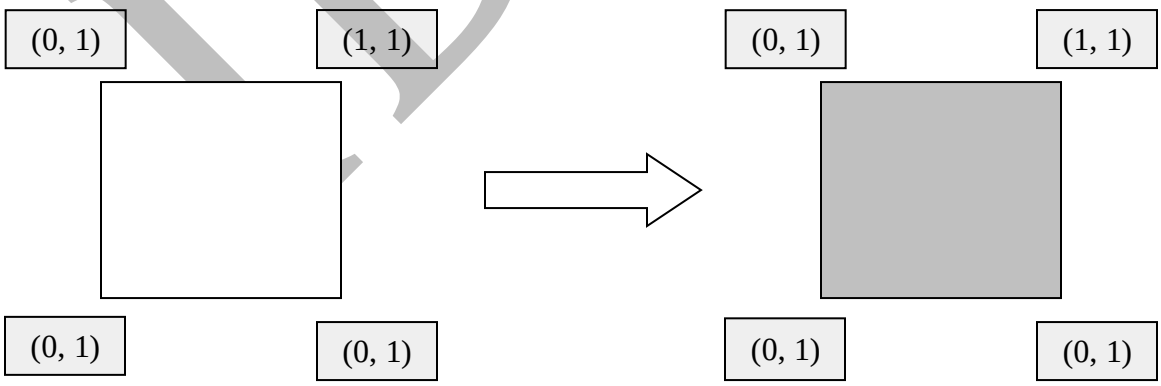
Each fuzzy unit indicates the degree to which the neuron belongs to the n-dimensional fuzzy set.The neuronal state pace (the set of all n-possibilities) equals the set of all n-dimensional fit vectors (the fuzzy power set), given by $I^n = [0, 1] * [0, 1] * \dots * [0, 1]$.This power set has 2^n vertices, which is an n-dimensional unit cube.

Non Fuzzy Set To Fuzzy Set: 1 Dimension

- Consider a non fuzzy set X = {x1}, contains only one element.
- The power set of X = {Ø, {x1}} where Ø = 0 and {x1} = 1, the binary bits.
- The corresponding fuzzy set contains all the values from 0 to 1.
- Thus,

Non Fuzzy Set To Fuzzy Set: 2 Dimensions

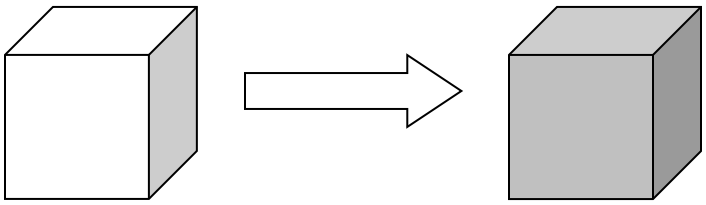
- Consider a non fuzzy set X = {x1, x2}, contains elements.
- The power set of X = {Ø, {x1}, {x2}, {x1, x2}} where Ø = (0, 0), {x1} = (1, 0), {x2} = (0, 1) and {x1, x2} = (1, 1).
- The points correspond to vertices of a unit square.
- Thus,



Non Fuzzy Set To Fuzzy Set: 3 Dimensions

- Consider a non fuzzy set X = {x1, x2, x3}, contains 3 elements.
- The power set of X = {Ø, {x1}, {x2}, {x3}, {x1, x2}, {x1, x3}, {x2, x3}, {x1, x2, x3}} where Ø = (0, 0, 0), {x1} = (1, 0, 0), {x2} = (0, 1, 0), {x3} = (0, 0, 1), {x1, x2} = (1, 1, 0), {x1, x3} = (1, 0, 1), {x2, x3} = (0, 1, 1) and {x1, x2, x3} = (1, 1, 1).

- The points correspond to vertices of a unit cube.
- Thus,



Non Fuzzy Set To Fuzzy Set: N Dimension

- In general, a n-vector non-fuzzy set value corresponds to an n-dimensional fit vector in the n-cube I^n = [0, 1] * [0, 1] * … * [0, 1].
- The mid point (1/2, 1/2… 1/2) of this n–cube corresponds to the paradoxes of logic where truth and falsity has same value.

Fit Vector: Example

- 1-dimension: 1/3
- 2-dimension: (1/3, 2/5) where m_A (x1) = 1/3 and m_A (x2) = 2/5.
- 3-dimension: (1/3, 2/5, 3/4) where m_A (x1) = 1/3, m_A (x2) = 2/5 and m_A (x3) = 3/4.

SUBSETHOOD THEOREM

Membership functions for fuzzy sets can be defined in any number of ways as long as they follow the rules of the definition of a fuzzy set. The Shape of the membership function used defines the fuzzy set and so the decision on which type to use is dependent on the purpose. The membership function choice is the subjective aspect of fuzzy logic, it allows the desired values to be interpreted appropriately.

- Subsethood measures the degree to which set A is a subset of B and is denoted by S(A, B).
- S(A, B) = Degree (A C B) = M(A ∩ B) / M(A) = P(B/A) where M(A) denotes the fuzzy of count of fit vector, i.e. if A = (a1, a2, …, an) the M(A) = a1 + a2 + … + an where 0 <= S(A, B) <= 1.
- Question: Apply subsethood theorem for R^3 with A = (3/4, 1/3, 1/6) and B = (1/4, 1/2, 1/3).
- Answer: X = R^3 where X = {x1, x2, x3}, contains 3 elements.
- The power set of X = {(0, 0, 0), (1, 0, 0), (0, 1, 0), (0, 0, 1), (1, 1, 0), (1, 0, 1), (0, 1, 1), (1, 1, 1)}
- Consider the fuzzy subset B = (1/4, 1/2, 1/3).
  If A and B are fuzzy sets, then m_A (A ∩ B) = min(m_A(A), m_A(B)) and m_A(A U B) = max(m_A (A), m_A (B)).
  (A ∩ B) = min(A, B) = (1/4, 1/3, 1/6)
  M(A ∩ B) = 1/4 + 1/3 + 1/6 = 3/4.
  M(A) = 3/4 + 1/3 + 1/6 = 5/4.

$S(A, B) = M(A \cap B) / M(A) = (3/4) / (5/4) = 3/5 = 60\%$.

Probability as Subsethood

- Consider a statistical experiment X of n trials.
- Suppose A defines the subset of successful trials.
- Let there be n_A successes out of n trials.
- Let 1 denote success and 0 denote failure.
- $S (A, X) = M (A \cap X) / M (X) = M (A) / M (X) = n_A / n = P (A)$.
- Thus probability reduces to subsethood.

Exercise:

1. Explain the concept of Fuzzy system with suitable example.
2. State the applications of fuzzy system.
3. Illustrate logic creates bivalent paradox.
4. Derived fuzzy as mid-point.
5. Explain in brief about degree of membership.
6. Explain in brief about subsethood theorem.

Chapter 9

Neural Networks

9.1 Introduction

A neural network can be defined as a computational model consisting of a network architecture of artificial neurons. This structure contains a set of parameters than can be adjusted to perform certain tasks. The brain contains *neurons* which are kind of like organic switches. These can change their output state depending on the strength of their electrical or chemical input. The neural network in a person's brain is a hugely interconnected network of neurons, where the output of any given neuron may be the input to thousands of other neurons. Learning occurs by repeatedly activating certain neural connections over others, and this reinforces those connections. This makes them more likely to produce a desired outcome given a specified input. This learning involves *feedback* – when the desired outcome occurs, the neural connections causing that outcome become strengthened.

9.1.1 BIOLOGICAL NEURAL SYSTEMS

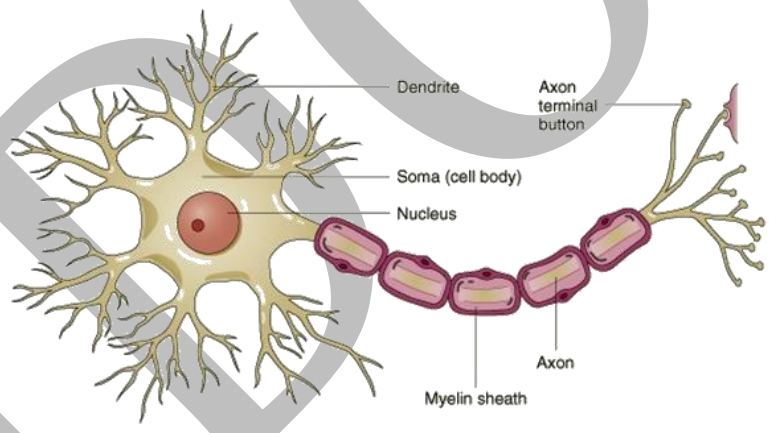


Figure 9.1: Biological Neural Network

The brain is composed of approximately 100 billion (1011) neurons.A typical neuron collects signals from other neurons through a host of fine structures called dendrites.The neuron sends out spikes of electrical activity through a long, thin strand known as an axon, which splits into thousands of branches.At the end of the branch, a structure called a synapse converts the activity from the axon into electrical effects that inhibit or excite activity in the connected neurons.When a neuron receives excitatory input that is sufficiently large compared with its inhibitory input, it sends a spike of electrical activity down its axon.Learning occurs by changing the effectiveness of the synapses so that the influence of one neuron on the other changes.

A typical neuron consists of the following four parts with the help of which we can explain its working –

- **Dendrites** − they are tree-like branches, responsible for receiving the information from other neurons it is connected to. In other sense, we can say that they are like the ears of neuron.
- **Soma** − It is the cell body of the neuron and is responsible for processing of information, they have received from dendrites.
- **Axon** − It is just like a cable through which neurons send the information.
- **Synapses** − It is the connection between the axon and other neuron dendrites.

ANN versus BNN

Before taking a look at the differences between Artificial Neural Network (ANN) and Biological Neural Network (BNN), let us take a look at the similarities based on the terminology between these two.

Biological Neural Network (BNN)	Artificial Neural Network (ANN)
Soma	Node
Dendrites	Input
Synapse	Weights or Interconnections
Axon	Output

The concerns with Neural Networks:

• Computer scientists want to find out about the properties of non-symbolic information processing with neural nets and about learning systems in general.
• Statisticians use neural nets as flexible, nonlinear regression and classification models.
• Engineers of many kinds exploit the capabilities of neural networks in many areas, such as signal processing and automatic control.
• Cognitive scientists view neural networks as a possible apparatus to describe models of thinking and consciousness (high-level brain function).
• Neuro-physiologists use neural networks to describe and explore medium-level brain function (e.g.memory, sensory system, motorics).
•Physicists use neural networks to model phenomena in statistical mechanics and for a lot of other tasks.
•Biologists use Neural Networks to interpret nucleotide sequences.
•Philosophers and some other people may also be interested in Neural Networks for various reasons.

9.1.2 Brain : A Dynamic System

The brain is a highly complex, non-linear, parallel information processing system. It performs tasks like pattern recognition, perception, and motor-control, many times faster than the fastest digital computers.Neurons have been modeled as nonlinear systems for decades now, but dynamical systems emerge in numerous other ways in the nervous system. We can duplicate working of brain in machines.These machines are expected to work smarter.This 'smartness' is referred as machine intelligence.Artificial Neural networks and Fuzzy systems are two adaptive machine intelligent systems.Dynamical neuroscience describes the non-linear dynamics at many

levels of the brain from single neural cells to cognitive processes, sleep states and the behavior of neurons in large-scale neuronal simulation.

9.2 Neural and Fuzzy Systems as Function Estimators

Neural Networks and Fuzzy Systems estimate I/O functions.Both are model free in I/O analysis and so the same architecture can be used for different problems.Both are trainable dynamical systems using sample data.These sample information is encoded in parallel – distributed framework.Neural Networks have the property of 'recognition without definition' and learn from previous experiences for e.g. child. This property helps in generalizing and better learning for e.g. Natural Language development. The Distributed encoding in Neural Networks helps in recognizing partial patterns, fault tolerance and graceful degradation. The Neural Networks contains a collection of processing units called neurons.Neurons work as I/O functions and synapses (joints) work as adjustable weights.Thus Neural Networks behave as adaptive function estimators.

9.2.1 Neural Networks as Trainable Dynamical Systems

- Network activity in Neural Network follows a trajectory in the state space of all possibilities.Each point in the state space is a possible Neural Network configuration.
- Corresponding to an input, trajectory begins and with a solution trajectory ends. E.g. pattern recognition.Here synaptic values gradually change to learn new patterns.

9.3 Artificial Neural Networks

Artificial Neural Network (ANN) is an efficient computing system whose central theme is borrowed from the analogy of biological neural networks. ANNs are also named as "artificial neural systems," or "parallel distributed processing systems," or "connectionist systems." ANN acquires a large collection of units that are interconnected in some pattern to allow communication between the units. These units, also referred to as nodes or neurons, are simple processors which operate in parallel. Every neuron is connected with other neuron through a connection link. Each connection link is associated with a weight that has information about the input signal. This is the most useful information for neurons to solve a particular problem because the weight usually excites or inhibits the signal that is being communicated. Each neuron has an internal state, which is called an activation signal as shoe in figure 9.2. Output signals, which are produced after combining the input signals and activation rule, may be sent to other units.

Artificial neurons are analogous to their biological inspirers.

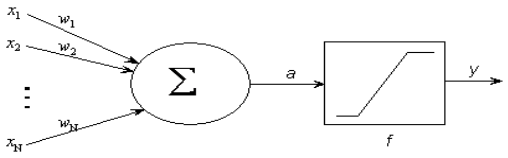


Figure 9.2: Artificial Neural network

Here the neuron is actually a processing unit, it calculates the weighted sum of the input signal to the neuron to generate the activation signal a, given by

$$a = \sum_{i=1}^{N} w_i x_i$$

Where, w_i is the strength of the synapse connected to the neuron, x_i is an input feature to the neuron. The activation signal is passed through a transform function to produce the output of the neuron, given by

$$y = f(a)$$

The transform function can be linear, or non-linear, such as a threshold or sigmoid function

9.4 Classification of Neural Network Models

We can classify Neural Networks based on whether they learn with supervision or whether they contain feedback. The neural network models are classified as feed forward and feedback.

Supervised learning:

It is called supervised learning because the process of a learning (from the training dataset) can be thought of as a teacher who is supervising the entire learning process. Thus, the "learning algorithm" iteratively makes predictions on the training data and is corrected by the "teacher", and the learning stops when the algorithm achieves an acceptable level of performance (or the desired accuracy).

Supervised learning is the learning of the model where with input variable (say, x) and an output variable (say, Y) and an algorithm to map the input to the output.

Y = f(X)

The basic aim is to approximate the mapping function(mentioned above) so well that when there is a new input data (x) then the corresponding output variable can be predicted.

Example of Supervised Learning

Suppose there is a basket which is filled with some fresh fruits, the task is to arrange the same type of fruits at one place.Also, suppose that the fruits are apple, banana, orange.

Suppose one already knows from their *previous work* (or experience) that, the shape of each and every fruit present in the basket so, it is easy for them to arrange the same type of fruits in one place.

Unsupervised learning is the training of machine using information that is neither classified nor labeled and allowing the algorithm to act on that information without guidance. Unsupervised learning is where only the input data (say, X) is present and no corresponding output variable is

there. The main aim of Unsupervised learning is to model the distribution in the data in order to learn more about the data.

Example of Unsupervised Learning

Suppose there is a basket filled with some fresh fruits. The task is to arrange the same type of fruits at one place.This time there is no information about those fruitsbeforehand, it's the first time that the fruits are being seen or discovered, how to group similar fruits without any prior knowledge about those.First, any physical characteristic of a particular fruit is selected. Suppose *color*.

Then the fruits are arranged on the basis of the color. The groups will be something as shown below:
RED COLOR GROUP: apples & cherry fruits.
GREEN COLOR GROUP: bananas & grapes.

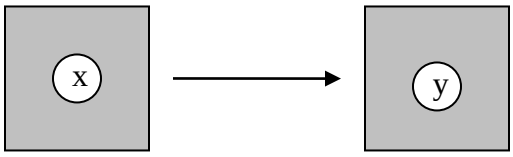
So now, take another physical character say, *size*, so now the groups will be something like this.
RED COLOR AND **BIG SIZE**: apple.
RED COLOR AND **SMALL SIZE**: cherry fruits.
GREEN COLOR AND **BIG SIZE**: bananas.
GREEN COLOR AND **SMALL SIZE**: grapes.

Supervised & Feed Forward Perceptron LMS Back Propagation	**Supervised & Feed Back** Recurrent Back Propagation
Unsupervised & Feed Forward Self-Organizing Map Data Clustering	**Unsupervised & Feed Back** Boltzmann Learning Hopfield Network

8.5 Intelligent Behavior as Adaptive Model Free Estimation

Intelligent systems adaptively functions from data without a model for I/O processing. The living creatures respond to stimuli, or, we map stimuli to responses (like f: X → Y).Intelligent systems associate similar responses with similar stimuli.They produce minor changes if the inputs are changed slightly. As illustrated in the following example let's say

S is stimuli and R as response in the spaces.
Consider balls Bx and By. Here f (Bx) = By.



For every similar y' in By, we can find some similar stimulus x' in Bx such that y' = f(x').That is, f is an onto map.

Intelligent systems are creative as discussed in the following example.
The measure of creativity of f is given by :
$C_{Bx}(f) = V(By) / V(Bx)$,

Where, V – volume.

- Case 1: $C_{Bx}(f) = 0$.
  =>$V(B_y) = 0$ or $V(B_x)$ = infinity.
  => $B_y = 0$ or $V(B_x)$ = infinity.
  =>f is a constant function or B_x is of infinite radius.
  =>f is "dull" or stimuli overwhelm responses.
- Case 2: $C_{Bx}(f)$ = infinity.
  => Infinite radius for B_y.
  =>f emits infinite responses.

Small variations in input provide simplest novel stimuli.This manifests as creativity.

9.6 Learning as Change

Intelligent systems learn or adapt.Learning or adaptation means just parameter change.
The parameter may be:Average transmission rate at synaptic junctions.Gene frequency at the locus of chromosome.Practically, learning means change.So, learning laws could describe a dynamic system.We can impart any system to encode or decode information.

- E.g. mowing of lawn of grass.
- Lawn = brain.

Supervised learning uses class – membership information.It can 'know' that 'belong' and 'not belong'.E.g. speech recognition system at airport – supervised using 'carrot and stick' policy.Unsupervised systems adaptively cluster, like patterns with like patterns.Biological synapses learn without supervision.

9.6.1 Expert System Knowledge as Rule Trees

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

The components of ES include −

- Knowledge Base
- Inference Engine
- User Interface

As discuss in the following figure

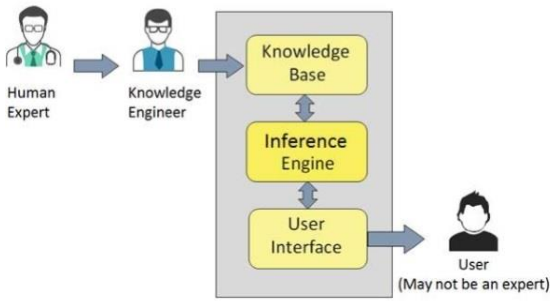


Figure 9.3 Expert System

Knowledge Base

It contains domain-specific and high-quality knowledge.Knowledge is required to exhibit intelligence. The success of any ES majorly depends upon the collection of highly accurate and precise knowledge.

What is Knowledge?

The data is collection of facts. The information is organized as data and facts about the task domain. **Data, information,** and **past experience** combined together are termed as knowledge.

Components of Knowledge Base

The knowledge base of an ES is a store of both, factual and heuristic knowledge.

- **Factual Knowledge** − It is the information widely accepted by the Knowledge Engineers and scholars in the task domain.
- **Heuristic Knowledge** − It is about practice, accurate judgement, one's ability of evaluation, and guessing.

Knowledge representation

It is the method used to organize and formalize the knowledge in the knowledge base. It is in the form of IF-THEN-ELSE rules.

Knowledge Acquisition

The success of any expert system majorly depends on the quality, completeness, and accuracy of the information stored in the knowledge base.

The knowledge base is formed by readings from various experts, scholars, and the **Knowledge Engineers**. The knowledge engineer is a person with the qualities of empathy, quick learning, and case analyzing skills.He acquires information from subject expert by recording, interviewing, and observing him at work, etc. He then categorizes and organizes the information in a meaningful way, in the form of

IF-THEN-ELSE rules, to be used by interference machine. The knowledge engineer also monitors the development of the ES.

Inference Engine

Use of efficient procedures and rules by the Inference Engine is essential in deducting a correct, flawless solution.In case of knowledge-based ES, the Inference Engine acquires and manipulates the knowledge from the knowledge base to arrive at a particular solution.

In case of rule based ES, it −

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

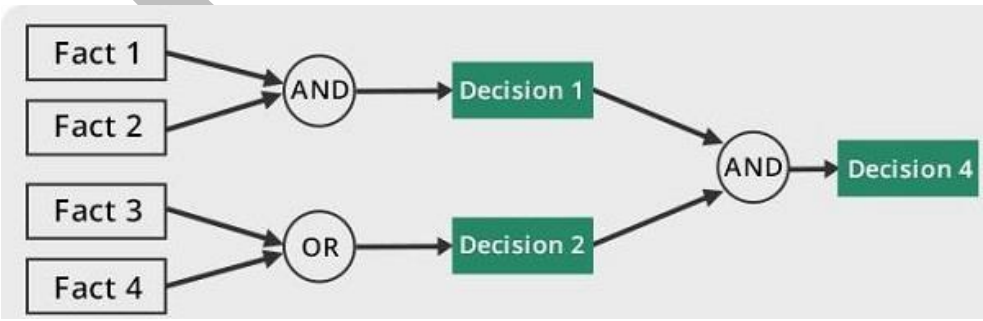
To recommend a solution, the Inference Engine uses the following strategies −

- Forward Chaining
- Backward Chaining

Forward Chaining

It is a strategy of an expert system to answer the question, **"What can happen next?"**Here, the Inference Engine follows the chain of conditions and derivations and finally deduces the outcome. It considers all the facts and rules, and sorts them before concluding to a solution.Forward chaining starts with some initial information and work forward, attempting to match that information with a rule.Once a fact has been matched to the IF part of the rule, the rule is fired.The action could produce new knowledge or a new fact that is stored in the knowledge base.This new fact may then be used to search out the next appropriate rule.This searching and matching process continues until a final conclusion rule is fired.

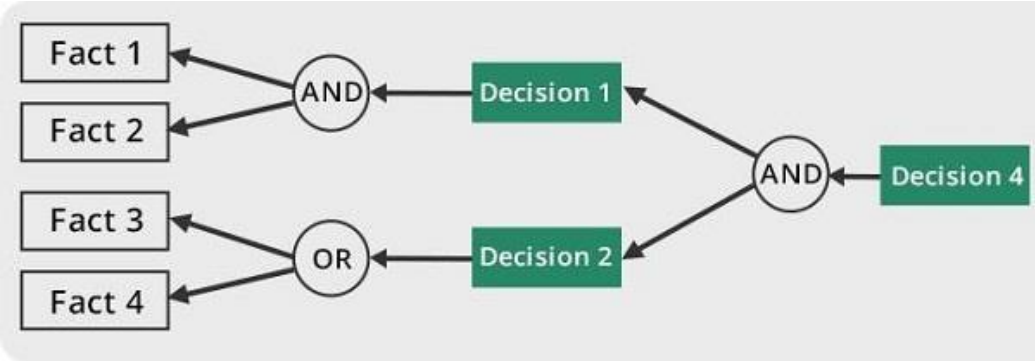
This strategy is followed for working on conclusion, result, or effect. For example, prediction of share market status as an effect of changes in interest rates.



Backward Chaining

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

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



Backward chaining starts with a fact in the database, but this time it is the hypothesis.The rule interpreter then begins examining the THEN parts of rules for a match.The inference engine searches for an evidence to support the hypothesis originally stated.If a match is found, the database is updated recording the conditions or premises that the rule stated as necessary for supporting the matched conclusion.The chaining process continues with the system repeatedly attempting to match the right hand side of the rule against the current status of the system.The corresponding IF sides of the rules matched are used to generate new intermediate hypotheses or goal states which are recorded in the database.Again this backward chaining continues until the hypothesis is proved.

The choice of inference strategy with either forward or backward chaining is determined by the design of the system and the nature of the problem being solved.In large systems with many rules, the forward chaining or data driven approach may be too slow, as it will generate many sequences of rules.The search, as a result, can go off on undesired directions exploring alternatives that do not fit in the problem.In such cases a backward chaining or goal driven approach may be advantageous.On the other hand, a backward chaining process could get a fixation on a particular hypothesis and continue to explore it though the data available to support it may not be there.The system does not know when to switch the emphasis or context to a more appropriate search sequence.Some expert systems incorporate both forward and backward chaining.This speeds up the process and ensures a solution. The concurrent forward-backward search rapidly converges on an answer.

8.6 Fuzzy Associative Memory (FAM)

Fuzzy associative memories are transformations of FAM map fuzzy sets to fuzzy sets, units cube to units cube. A kind of content-addressable **memory** in which the recall occurs correctly if input data fall within a specified window consisting of an upper bound and a lower bound of the stored patterns. A **Fuzzy Associative Memory** is identified by a matrix of **fuzzy** values. It allows to map an input **fuzzy** set into an output **fuzzy** one.*Fuzzy associative memories* (FAMs) belong to the class of *fuzzy neural*

networks (FNNs). A FNN is an *artificial neural network* (ANN) whose input patterns, output patterns, and/or connection weights are fuzzy-valued.

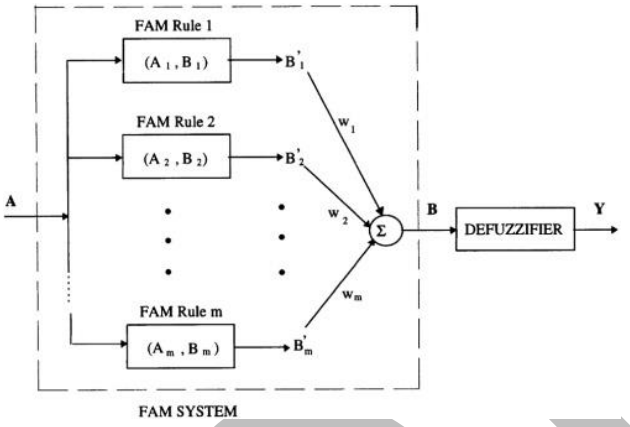


Figure 9.4: FAM architecture

9.6.1 Fuzzy Systems as Structured Numeric Estimators

Fuzzy systems encode structured knowledge in a numeric framework. For example We can enter a fuzzy association like (TALL, HEIGHT) as an entry in FAM (Fuzzy Associative Memory) rule matrix.

Let Θ – angle of pendulum, δΘ – angular velocity of pendulum, v – current to the motor control that adjusts pendulum.

All variables are fuzzy and v is the output and others are inputs.

Each variable has 5 fuzzy set values:

- Negative Medium (NM)
- Negative Small (NS)
- Zero (ZE)
- Positive Small (PS)
- Positive Medium (PM)

Fam Rules Of Pendulum

		Θ				
δΘ		NM	NS	ZE	PS	PM
	NM			PM		
	NS			PS		
	ZE	PM	PS	ZE	NS	NM
	PS			NS		
	PM			NM		

Usually fuzzy set values are defined as trapezoids.E.g. angle value 0 belongs to fuzzy value ZE to degree 1.The angle value 3 may belong to ZE only to degree say 0.6

9.6.2 Fuzzy and Neural Network as Function Estimator

Neural and fuzzy systems **estimate** sampled functions and behave as associative memories. They share a key advantage over traditional statistical-estimation and adaptive control approaches to function estimation. They are *model-free* estimators. Neural and fuzzy systems estimate a function without requiring a mathematical description of how the output functionally depends on the input. They "learn from example". Neural and fuzzy systems differ in how they estimate sampled functions. They differin the kind of samples used, how they represent and store those samples.
Fuzzy systems estimate functions with *fuzzy set* samples (A_i, B_i). Neural systems use *numerical point* samples (xi, *yi)*. Both kinds of samples are from the input-output product space $X \times Y$. Figure 9.5illustrates the geometry of fuzzy-set and numerical-point samples taken from the function *f: X ->Y.*The fuzzy-set association (A_i, B_i) is sometimes called a *"rule."* T

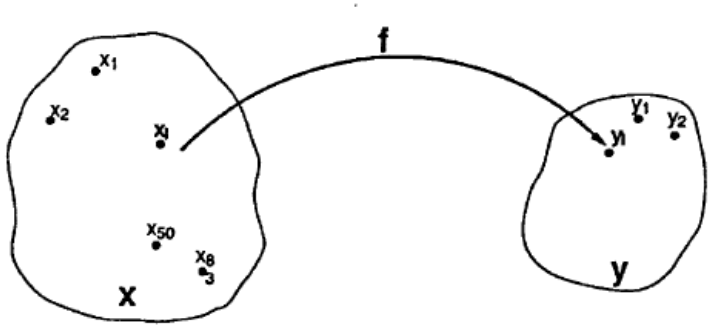


Figure 9.5: Function f maps X to range Y

9.6.3 Fuzzy Systems as Parallel Associators

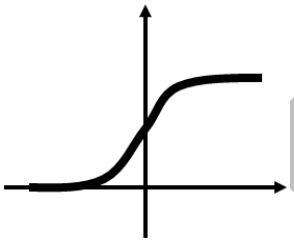
- Fuzzy Systems store and process FAM rules in parallel.
- $B = \Sigma W_J B_J$
- Adaptive Fuzzy Systems use sample data and neural / statistical algorithms to choose the coefficients.
- If the input fuzzy system define points in the unit hypercube I^n and output fuzzy system define points in the unit hypercube I^p then the transformation S defines a Fuzzy System: if S: $I^n \rightarrow I^p$
- S defines an adaptive Fuzzy System if it changes with time i.e. dS/dt != 0.

9.6.4 Fuzzy Systems as Principle Based Systems

AI expert systems work through rules.Inference is performed by traversing through the decision tree.The tree can be shallow or deep.E.g. shallow – chess, deep – water jug.Shallow tree use only a small proportion of the stored knowledge in the inference.In that sense, they are non-interactive.Fuzzy systems are shallow but interactive.Every inference fires every FAM rule to some degree.AI expert systems use rule based approach.But fuzzy systems use principle based approach.E.g. AI vs. fuzzy judge.Rules apply "in an all-or-none" fashion.Principles have a dimension of weight or importance.Principles evolve, but rules are static.Adaptive Fuzzy Systems use neural techniques to abstract fuzzy principles from samples.This is similar to our acquisition of knowledge.

9.7 NEURONS AS FUNCTIONS

An activation function is a very important feature of an artificial neural network, they basically decide whether the neuron should be activated or not. Activation functions make the back-propagation possible since the gradients are supplied along with the error to update the weights and biases. Activation function decides, whether a neuron should be activated or not by calculating weighted sum and further adding bias with it. The purpose of the activation function is to **introduce non-linearity** into the output of a neuron.

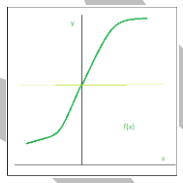


In this, we consider a threshold value and if the value of net input say **y** is greater than the threshold then the neuron is activated.

$$f(x) = 1\,, if\ x >= 0$$
$$f(x) = 0\,, if\ x < 0$$

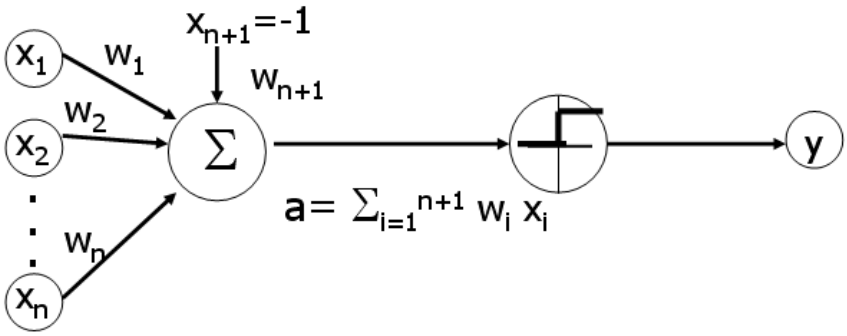
Sigmoid Function:

Sigmoid function is a widely used activation function. It is defined as:



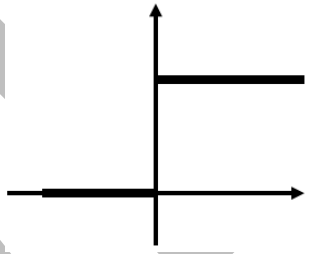
This is a smooth function and is continuously differentiable. The biggest advantage that it has over step and linear function is that it is non-linear. This is an incredibly cool feature of the sigmoid function.

Effect of Signal Function

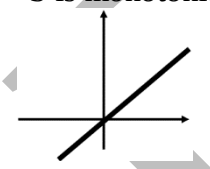


- Where $\Theta = w_{n+1}$
- Usually signal functions are monotone non-decreasing i.e. dS/dt>= 0.
- But, dS/dt = (dS/dx) (dx/dt) i.e. signal velocity depends on activation velocity.

- Logistical signal function
  - $S(x) = 1 / (1 + e^{-cx})$
  - $S' = c\ S\ (1 - S)$
  - $S' > 0$
  - => S is monotonic increasing.
- Hyperbolic – tangent signal function
  - $S(x) = \tanh(c\ x)$
  - $S' = c\ (1 - S^2)$
  - $S' > 0$
  - => S is monotonic increasing.
- Threshold linear signal function
  - It is binary function.
  - S(x) = 1, if c x >= 1
  - = 0, if c x < 0
  - = c x, otherwise
  - S' = c
  - => S' > 0
  - => S is monotonic increasing.



- Linear signal function
  - S(x) = c x
  - S' = c
  - => S' > 0
  - => S is monotonic increasing.



- Threshold exponential signal function
  - $S(x) = \min(1, e^{cx})$
  - $S' = c\, e^{cx}$, if $e^{cx} < 1$
  - $\Rightarrow S' > 0$
  - $\Rightarrow$ S is monotonic increasing.
- Exponential distribution signal function
  - $S(x) = \max(0, 1 - e^{-cx})$
  - $S' = c\, e^{-cx}$, if $x > 0$
  - $\Rightarrow S' > 0$
  - $\Rightarrow$ S is monotonic increasing.
  - And $S'' = c^2\, e^{-cx} < 0$
  - $\Rightarrow$ Strictly convex.
- Ratio polynomial signal function
  - $S(x) = \max(0, (x^n / (c + x^n))$
  - $S' = c\, n\, x^{n-1} / (c + x^n)^2$, $x > 0$
  - $\Rightarrow S' > 0$

  $\Rightarrow$ S is monotonic increasing

Exercise:

1. Explain the working mechanism of biological neural network architecture.
2. What is artificial neural network? Explain with suitable example.
3. Give comparison between biological neural network and ANN.
4. Explain neural network as function estimator.
5. Give comparison between supervised and unsupervised learning.
6. Give comparison between forward and backward chaining.
7. Write a note on FAM architecture.
8. Explain different neuron activation functions.

Chapter 10
A Gentle Introduction to Genetic Algorithms

10.1 Introduction

Every organism has a set of rules, a blueprint so to speak, describing how that organism is built up from the tiny building blocks of life. These rules are encoded in the genes of an organism, which in turn are connected together into long strings called chromosomes. Each gene represents a specific trait of the organism, like eye colour or hair colour, and has several different settings. For example, the settings for a hair colour gene may be blonde, black or auburn. These genes and their settings are usually referred to as an organism's genotype. The physical expression of the genotype - the organism itself - is called the phenotype. When two organisms mate they share their genes. The resultant offspring may end up having half the genes from one parent and half from the other. This process is called recombination. Very occasionally a gene may be mutated. Normally this mutated gene will not affect the development of the phenotype but very occasionally it will be expressed in the organism as a completely new trait.

Life on earth has evolved to be as it is through the processes of natural selection, recombination and mutation. Genetic Algorithms are search and optimization techniques based on Darwin's Principle of Natural Selection.
Darwin's Principle Of Natural Selection states that if there are organisms that reproduce, and if offspring's inherit traits from their progenitors, and if there is variability of traits, and if the environment cannot support all members of a growing population, then those members of the population with less-adaptive traits (determined by the environment) will die out, and then those members with more-adaptive traits (determined by the environment) will thrive. The result is the evolution of species.

Evolution

- The context of evolution is a population (of organisms, objects, agents ...) that survives for a limited time (usually) and then dies.
- Some produce offspring for succeeding generations, the 'fitter' ones tend to produce more.
- Over many generations, the make-up of the population changes.
- Without the need for any individual to change, successive generations, the 'species' changes, in some sense (usually) adapts to the conditions.

10.2 What are Genetic Algorithms?

Nature has always been a great source of motivation to all mankind. Genetic Algorithms (GAs) are search based algorithms based on the concepts of natural selection and genetics. GAs are a subset of a much larger branch of computation known as **Evolutionary Computation**.

GAs were developed by John Holland and his students and colleagues at the University of Michigan, most notably David E. Goldberg and has since been tried on various optimization problems with a high degree of success.

In GAs, we have a **pool or a population of possible solutions** to the given problem. These solutions then undergo recombination and mutation (like in natural genetics), producing new children, and the process is repeated over various generations. Each individual (or candidate solution) is assigned a fitness value (based on its objective function value) and the fitter individuals are given a higher chance to mate and yield more "fitter" individuals. This is in line with the Darwinian Theory of "Survival of the Fittest".

In this way we keep "evolving" better individuals or solutions over generations, till we reach a stopping criterion.

Genetic Algorithms are sufficiently randomized in nature, but they perform much better than random local search (in which we just try various random solutions, keeping track of the best so far), as they exploit historical information as well.

Advantages of GAs

GAs have various advantages which have made them immensely popular. These include –

- Does not require any derivative information (which may not be available for many real-world problems).
- Is faster and more efficient as compared to the traditional methods.
- Has very good parallel capabilities.
- Optimizes both continuous and discrete functions and also multi-objective problems.
- Provides a list of "good" solutions and not just a single solution.
- Always gets an answer to the problem, which gets better over the time.
- Useful when the search space is very large and there are a large number of parameters involved.

Limitations of GAs

Like any technique, GAs also suffer from a few limitations. These include −

- GAs are not suited for all problems, especially problems which are simple and for which derivative information is available.
- Fitness value is calculated repeatedly which might be computationally expensive for some problems.
- Being stochastic, there are no guarantees on the optimality or the quality of the solution.
- If not implemented properly, the GA may not converge to the optimal solution.

10.3 Genetic Algorithm – Beginning

Genetic Algorithms have the ability to deliver a “good-enough” solution “fast-enough”. This makes genetic algorithms attractive for use in solving optimization problems. The reasons why GAs are needed are as follows −

Solving Difficult Problems

In computer science, there is a large set of problems, which are **NP-Hard**. What this essentially means is that, even the most powerful computing systems take a very long time (even years!) to solve that problem. In such a scenario, GAs prove to be an efficient tool to provide **usable near-optimal solutions** in a short amount of time.

10.3.1 Failure of Gradient Based Methods

Traditional calculus based methods work by starting at a random point and by moving in the direction of the gradient, till we reach the top of the hill. This technique is efficient and works very well for single-peaked objective functions like the cost function in linear regression. But, in most real-world situations, we have a very complex problem called as landscapes, which are made of many peaks and many valleys, which causes such methods to fail, as they suffer from an inherent tendency of getting stuck at the local optima as shown in the following figure.

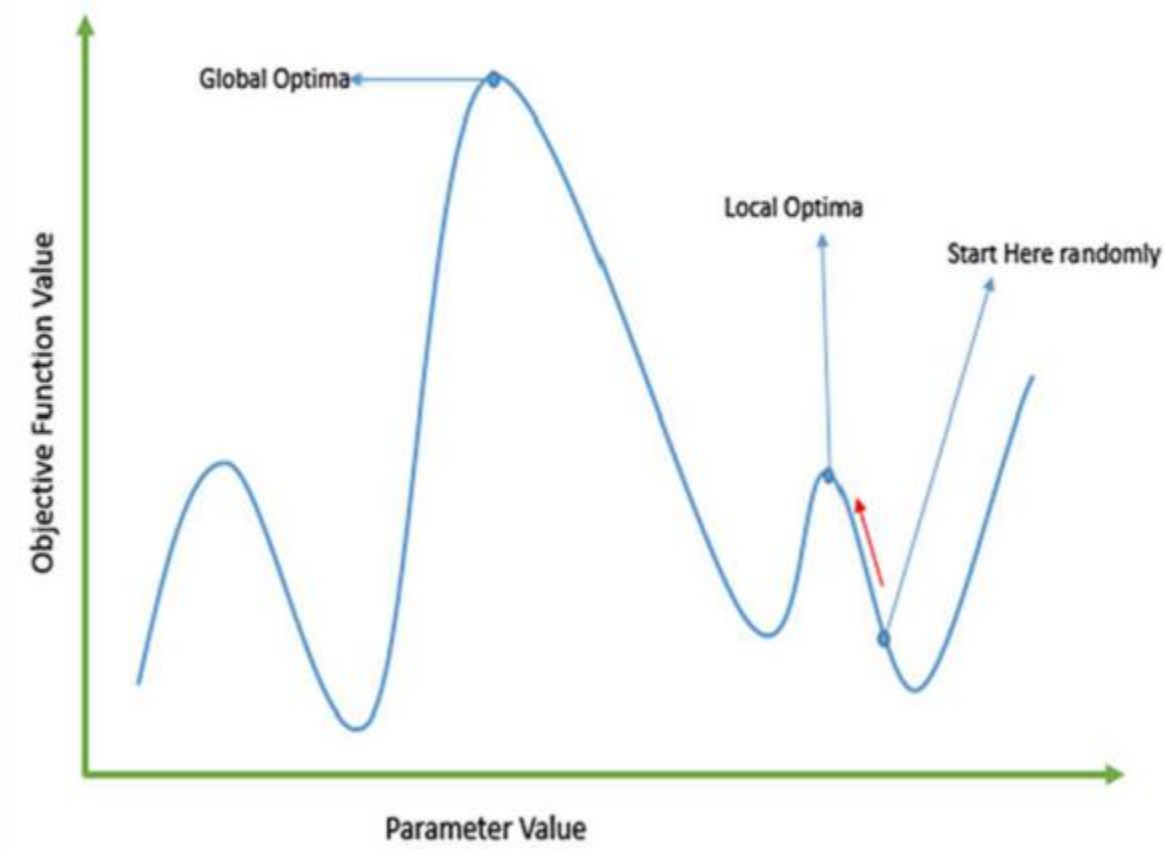


Figure 10.1: Performance of gradient based method

Getting a Good Solution Fast

Some difficult problems like the Travelling Salesperson Problem (TSP), have real-world applications like path finding and VLSI Design. Now imagine that you are using your GPS Navigation system, and it takes a few minutes (or even a few hours) to compute the "optimal" path from the source to destination. Delay in such real world applications is not acceptable and therefore a "good-enough" solution, which is delivered "fast" is what is required.

10.4 Basic Terminology

Before beginning a discussion on Genetic Algorithms, it is essential to be familiar with some basic terminology which will be used throughout this tutorial.

- **Population** − It is a subset of all the possible (encoded) solutions to the given problem. The population for a GA is analogous to the population for human beings except that instead of human beings, we have Candidate Solutions representing human beings.
- **Chromosomes** − A chromosome is one such solution to the given problem.
- **Gene** − A gene is one element position of a chromosome.
- **Allele** − It is the value a gene takes for a particular chromosome.

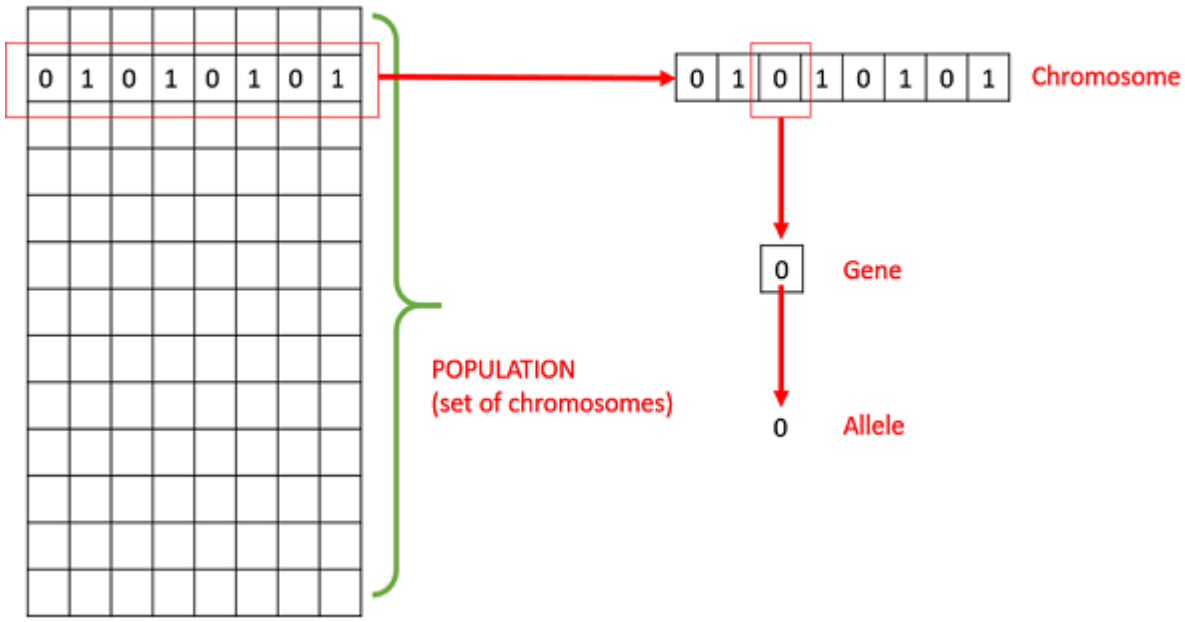


Figure 10.2: Genotype

- **Genotype** − Genotype is the population in the computation space. In the computation space, the solutions are represented in a way which can be easily understood and manipulated using a computing system.

- **Phenotype** − Phenotype is the population in the actual real world solution space in which solutions are represented in a way they are represented in real world situations.
- **Decoding and Encoding** − For simple problems, the **phenotype and genotype** spaces are the same. However, in most of the cases, the phenotype and genotype spaces are different. Decoding is a process of transforming a solution from the genotype to the phenotype space, while encoding is a process of transforming from the phenotype to genotype space. Decoding should be fast as it is carried out repeatedly in a GA during the fitness value calculation.

  For example, consider the 0/1 Knapsack Problem. The Phenotype space consists of solutions which just contain the item numbers of the items to be picked.

  However, in the genotype space it can be represented as a binary string of length n (where n is the number of items). A **0 at position x** represents that **x^{th}** item is picked while a 1 represents the reverse. This is a case where genotype and phenotype spaces are different.

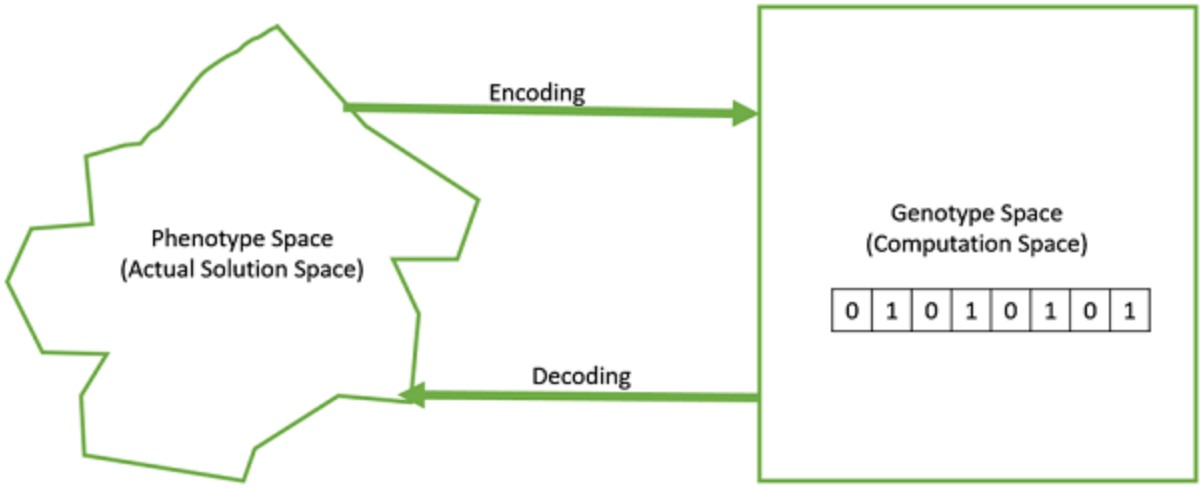


Figure 10.3: Encoding / decoding of phenotype and genotype

- **Fitness Function** − A fitness function simply defined is a function which takes the solution as input and produces the suitability of the solution as the output. In some cases, the fitness function and the objective function may be the same, while in others it might be different based on the problem.
- **Genetic Operators** − These alter the genetic composition of the offspring. These include crossover, mutation, selection, etc.

10.5 Basic Structure

Genetic Algorithm has its basic structure which is discuss as follows:

We start with an initial population (which may be generated at random or seeded by other heuristics), select parents from this population for mating. Apply crossover and mutation operators on the parents to generate new off-springs. And finally these off-springs replace the existing individuals in the population

and the process repeats. In this way genetic algorithms actually try to simulate the human evolution to some extent. The detail steps are discussed as follows:

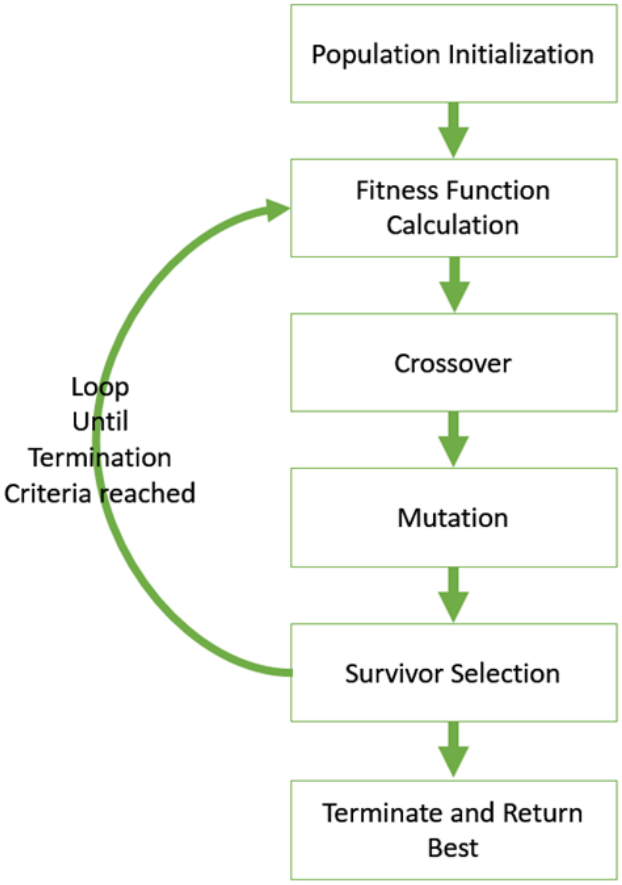


Figure 10.6: Basic structure of genetic algorithm

A generalized pseudo-code for a GA is explained in the following program −

```
GA()
   initialize population
   find fitness of population

   while (termination criteria is reached) do
      parent selection
      crossover with probability pc
      mutation with probability pm
      decode and fitness calculation
      survivor selection
      find best
   return best
```

From the above pseudocode it has been observed that improper representation can lead to poor performance of the GA.

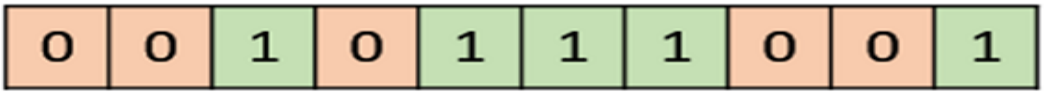
Therefore, choosing a proper representation, having a proper definition of the mappings between the phenotype and genotype spaces is essential for the success of a Genetic Algorithm.

In this section, some of the most commonly used representations for genetic algorithms are discussed. However, representation is highly problem specific and the reader might find that another representation or a mix of the representations mentioned here might suit his/her problem better.

1. Binary Representation

This is one of the simplest and most widely used representation in GAs. In this type of representation the genotype consists of bit strings.

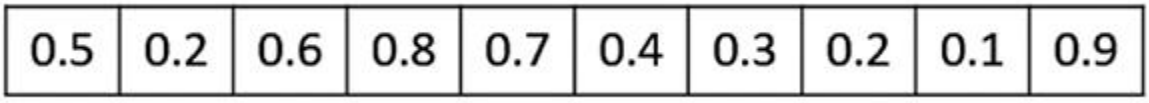
For some problems when the solution space consists of Boolean decision variables – yes or no, the binary representation is natural. Take for example the 0/1 Knapsack Problem. If there are n items, we can represent a solution by a binary string of n elements, where the x^{th} element tells whether the item x is picked (1) or not (0).



For other problems, specifically those dealing with numbers, we can represent the numbers with their binary representation. The problem with this kind of encoding is that different bits have different significance and therefore mutation and crossover operators can have undesired consequences. This can be resolved to some extent by using **Gray Coding,** as a change in one bit does not have a massive effect on the solution.

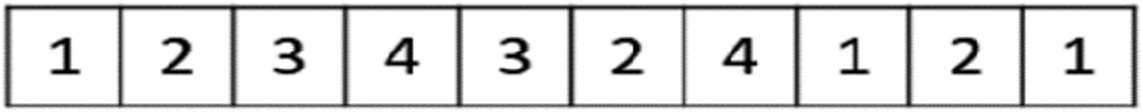
2. Real Valued Representation

For problems where we want to define the genes using continuous rather than discrete variables, the real valued representation is the most natural. The precision of these real valued or floating point numbers is however limited to the computer.



3. Integer Representation

For discrete valued genes, we cannot always limit the solution space to binary 'yes' or 'no'. For example, if we want to encode the four distances – North, South, East and West, we can encode them as **{0,1,2,3}**. In such cases, integer representation is desirable.



4. Permutation Representation

In many problems, the solution is represented by an order of elements. In such cases permutation representation is the most suited.

A classic example of this representation is the travelling salesman problem (TSP). In this the salesman has to take a tour of all the cities, visiting each city exactly once and come back to the starting city. The total distance of the tour has to be minimized. The solution to this TSP is naturally an ordering or permutation of all the cities and therefore using a permutation representation makes sense for this problem.

Population is a subset of solutions in the current generation. It can also be defined as a set of chromosomes. There are several things to be kept in mind when dealing with GA population −

- The diversity of the population should be maintained otherwise it might lead to premature convergence.
- The population size should not be kept very large as it can cause a GA to slow down, while a smaller population might not be enough for a good mating pool. Therefore, an optimal population size needs to be decided by trial and error.

The population is usually defined as a two dimensional array of – **size population, size x, chromosome size**.

10.5.1 Population Initialization

There are two primary methods to initialize a population in a GA. They are −

- **Random Initialization** − Populate the initial population with completely random solutions.
- **Heuristic initialization** − Populate the initial population using a known heuristic for the problem.

It has been observed that the entire population should not be initialized using a heuristic, as it can result in the population having similar solutions and very little diversity. It has been experimentally observed that the random solutions are the ones to drive the population to optimality. Therefore, with heuristic initialization, we just seed the population with a couple of good solutions, filling up the rest with random solutions rather than filling the entire population with heuristic based solutions.

It has also been observed that heuristic initialization in some cases, only effects the initial fitness of the population, but in the end, it is the diversity of the solutions which lead to optimality.

10.5.2 Population Models

There are two population models widely in use −

Steady State

In steady state GA, we generate one or two off-springs in each iteration and they replace one or two individuals from the population. A steady state GA is also known as **Incremental GA**.

Generational

In a generational model, we generate 'n' off-springs, where n is the population size, and the entire population is replaced by the new one at the end of the iteration. Basic Idea of Principle Of Natural Selection - "Select The Best, Discard The Rest".

The fitness function simply defined is a function which takes a **candidate solution to the problem as input and produces as output** how "fit" our how "good" the solution is with respect to the problem in consideration.

Calculation of fitness value is done repeatedly in a GA and therefore it should be sufficiently fast. A slow computation of the fitness value can adversely affect a GA and make it exceptionally slow.

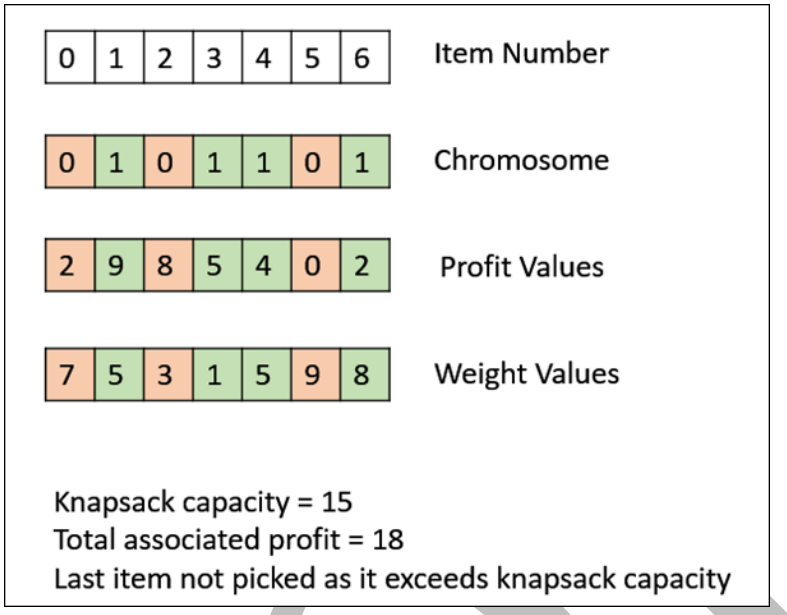
In most cases the fitness function and the objective function are the same as the objective is to either maximize or minimize the given objective function. However, for more complex problems with multiple objectives and constraints, an **Algorithm Designer** might choose to have a different fitness function.

A fitness function should possess the following characteristics −

- The fitness function should be sufficiently fast to compute.
- It must quantitatively measure how fit a given solution is or how fit individuals can be produced from the given solution.

In some cases, calculating the fitness function directly might not be possible due to the inherent complexities of the problem at hand. In such cases, we do fitness approximation to suit our needs.

The following image shows the fitness calculation for a solution of the 0/1 Knapsack. It is a simple fitness function which just sums the profit values of the items being picked (which have a 1), scanning the elements from left to right till the knapsack is full.

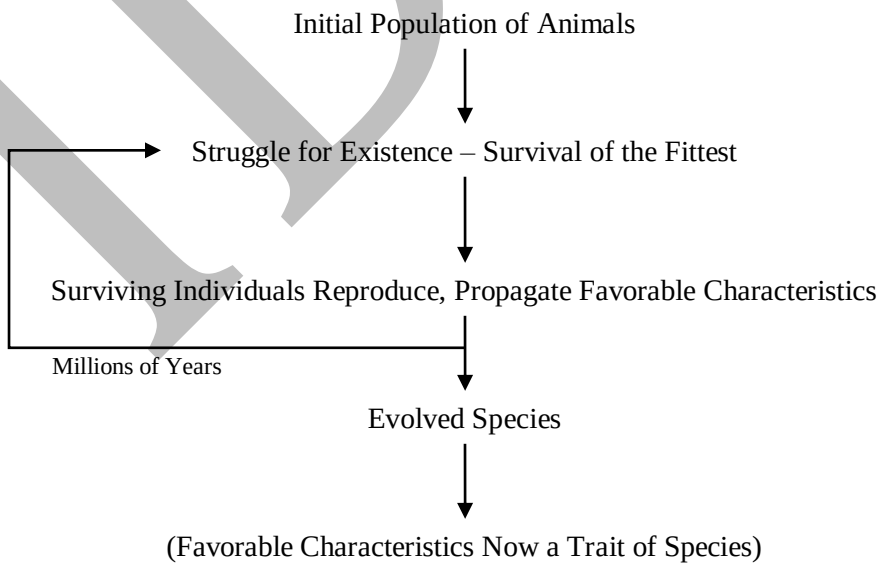


10.6 Evolution through Natural Selection

An example of natural selection. Giraffes have long necks. Giraffes with slightly longer necks could feed on leaves of higher branches when all lower ones had been eaten off:

- They had a better chance of survival.
- Favorable characteristic propagated through generations of giraffes.

Now, evolved species has long necks. Longer necks may have been a deviant characteristic (mutation) initially but since it was favorable, was propagated over generations. Now an established trait. So, some mutations are beneficial.



10.7 **Classes of Search Techniques**

A genetic algorithm (GA) is great for finding solutions to complex search problems. They're often used in fields such as engineering to create incredibly high quality products thanks to their ability to search a through a huge combination of parameters to find the best match. For example, they can search through different combinations of materials and designs to find the perfect combination of both which could result in a stronger, lighter and overall, better final product. They can also be used to design computer algorithms, to schedule tasks, and to solve other optimization problems. Genetic algorithms are based on the process of evolution by natural selection which has been observed in nature. They essentially replicate the way in which life uses evolution to find solutions to real world problems. Surprisingly although genetic algorithms can be used to find solutions to incredibly complicated problems, they are themselves pretty simple to use and understand. The basic process for a genetic algorithm is:

1. Initialization - Create an initial population. This population is usually randomly generated and can be any desired size, from only a few individuals to thousands.

2. Evaluation - Each member of the population is then evaluated and we calculate a 'fitness' for that individual. The fitness value is calculated by how well it fits with our desired requirements. These requirements could be simple, 'faster algorithms are better', or more complex, 'stronger materials are better but they shouldn't be too heavy'.

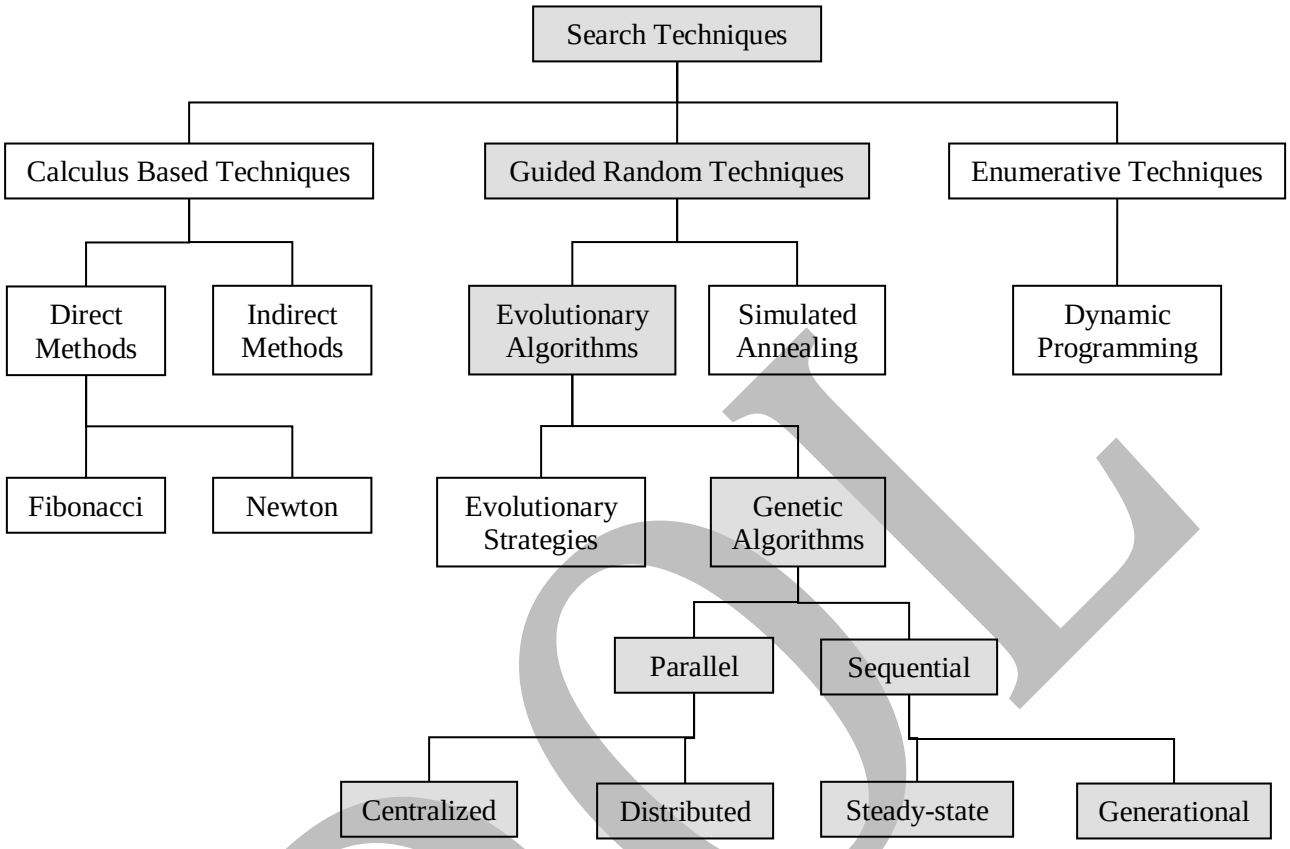
3. Selection - We want to be constantly improving our populations overall fitness. Selection helps us to do this by discarding the bad designs and only keeping the best individuals in the population. There are a few different selection methods but the basic idea is the same, make it more likely that fitter individuals will be selected for our next generation.

4. Crossover - During crossover we create new individuals by combining aspects of our selected individuals. We can think of this as mimicking how sex works in nature. The hope is that by combining certain traits from two or more individuals we will create an even 'fitter' offspring which will inherit the best traits from each of its parents.

5. Mutation - We need to add a little bit randomness into our populations' genetics otherwise every combination of solutions we can create would be in our initial population. Mutation typically works by making very small changes at random to an individual’s genome.

6. And repeat! - Now we have our next generation we can start again from step two until we reach a termination condition.

The population of individuals are maintained within search space. Each individual represent a solution in search space for given problem.



Evolutionary algorithm is a subset of evolutionary computation. Evolutionary computation is a general term for several computational techniques. Evolutionary computation represents powerful search and optimization paradigm influenced by biological mechanisms of evolution. Evolutionary algorithm refers to evolutionary computational models using randomness and genetic inspired operations. Evolutionary algorithms involve selection, recombination, random variation and competition of the individuals in a population of adequately represented potential solutions. The candidate solutions are referred as chromosomes or individuals.

Genetic Algorithm represent main paradigm of Evolutionary Computation.

- They simulate natural evolution, mimicking processes the nature uses such as Selection, Crossover, Mutation and Accepting.
- Genetic algorithm's simulate the survival of the fittest among individuals over consecutive generation for solving a problem.

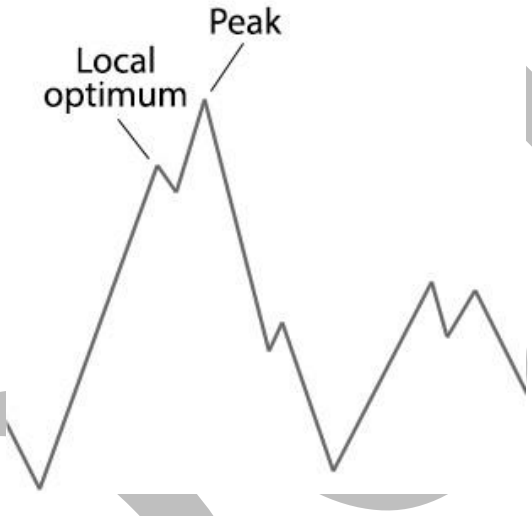
Termination

There are a few reasons why you would want to terminate your genetic algorithm from continuing it's search for a solution. The most likely reason is that your algorithm has found a solution which is good

enough and meets a predefined minimum criteria. Offer reasons for terminating could be constraints such as time or money.

Limitations

Imagine you were told to wear a blindfold then you were placed at the bottom of a hill with the instruction to find your way to the peak. You're only option is to set off climbing the hill until you notice you're no longer ascending anymore. At this point you might declare you've found the peak, but how would you know? In this situation because of your blindfolded you couldn't see if you're actually at the peak or just at the peak of smaller section of the hill. We call this a local optimum. Below is an example of how this local optimum might look:



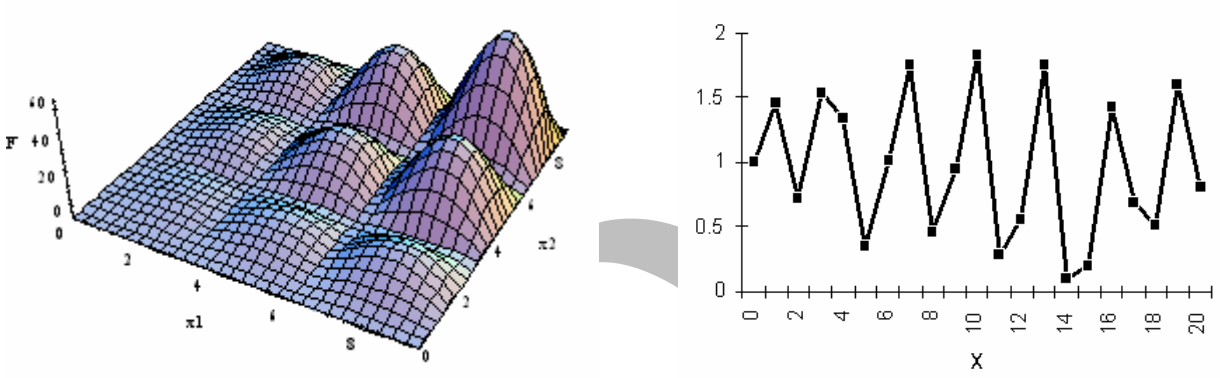
Unlike in our blindfolded hill climber, genetic algorithms can often escape from these local optimums if they are shallow enough. Although like our example we are often never able to guarantee that our genetic algorithm has found the global optimum solution to our problem. For more complex problems it is usually an unreasonable exception to find a global optimum, the best we can do is hope for is a close approximation of the optimal solution.

Search Methods

Directed search algorithms are based on the mechanics of biological evolution. It was developed by John Holland, University of Michigan (1970's). It is designed on the background of the adaptive processes of natural systems. In order to design an artificial systems software that retains the robustness of natural systems. It also provide efficient, effective techniques for optimization and machine learning applications. They are widely – used today in business, scientific and engineering circles.

- Blind random search does not use acquired information in deciding on the future direction of the search. A ***blind search*** (also called an ***uninformed search***) is a search that has no information about its domain. The only thing that a blind search can do is distinguish a non-goal state from a goal state.

- Hill climbing and gradient descent use acquired information; however, they are prone to becoming trapped on local optima. H**ill 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 to find a better solution by making an incremental change to the solution. If the change produces a better solution, another incremental change is made to the new solution, and so on until no further improvements can be found.



Gradient descent method is a way to find a local minimum of a function. The way it works is we start with an initial guess of the solution and we take the gradient of the function at that point. We **step the solution in the negative direction of the gradient** and we repeat the process. The algorithm will eventually converge where the gradient is zero (which correspond to a local minimum). Its brother, the gradient ascent, finds the local maximum nearer the current solution by stepping it towards the positive direction of the gradient. They are both first-order algorithms because they take only the first derivative of the function.

Exercise:

1. Explain the concept of genetic algorithms with suitable example.
2. State the advantages and dis-advantages of genetic algorithms.
3. Explain basic terminologies of genetic algorithms.
4. Explain the basic structure of genetic algorithm.
5. Explain different types of value presentations.
6. Explain in brief about search optimization methods.

Chapter 11

Genetic Algorithm Implementation

11.1 Introduction

The idea behind GA´s is to extract optimization strategies nature uses successfully - known as *Darwinian Evolution* - and transform them for application in mathematical optimization theory to find the global optimum in a defined *phase space*.

One could imagine a population of individual *"explorers"* sent into the optimization *phase-space*. Each explorer is defined by its genes, what means, its position inside the phase-space is coded in his genes. Every explorer has the duty to find a value of the quality of his position in the phase space. (Consider the phase-space being a number of variables in some technological process, the value of quality of any position in the phase space - in other words: any set of the variables - can be expressed by the yield of the desired chemical product.) Then the struggle of "life" begins. The three fundamental principles are

1. **Selection**
2. **Mating/Crossover**
3. **Mutation**

Only explorers (= genes) sitting on the best places will reproduce and *create a new population*. This is performed in the second step (*Mating/Crossover*). The "hope" behind this part of the algorithm is, that *"good"* sections of two parents will be recombined to yet *better* fitting children. In fact, many of the created children will not be successful (as in biological evolution), but a few children will indeed fulfill this hope. These "good" sections are named in some publications as building blocks.

Now there appears a problem. Repeating these steps, no *new* area would be *explored*. The two former steps would only *exploit* the already known regions in the phase space, which could lead to premature convergence of the algorithm with the consequence of missing the *global optimum* by exploiting some *local optimum*. The third step - the Mutation ensures the necessary accidental effects. One can imagine the new population being mixed up a little bit to bring some new information into this set of genes. Off course this has to happen in a well-balanced way!

Whereas in *biology* a gene is described as a macro-molecule with four different bases to code the *genetic information*, a *gene* in genetic algorithms is usually defined as a *bitstring* (a sequence of b 1´s and 0´s).

11.2 Working Mechanism of Genetic Algorithms

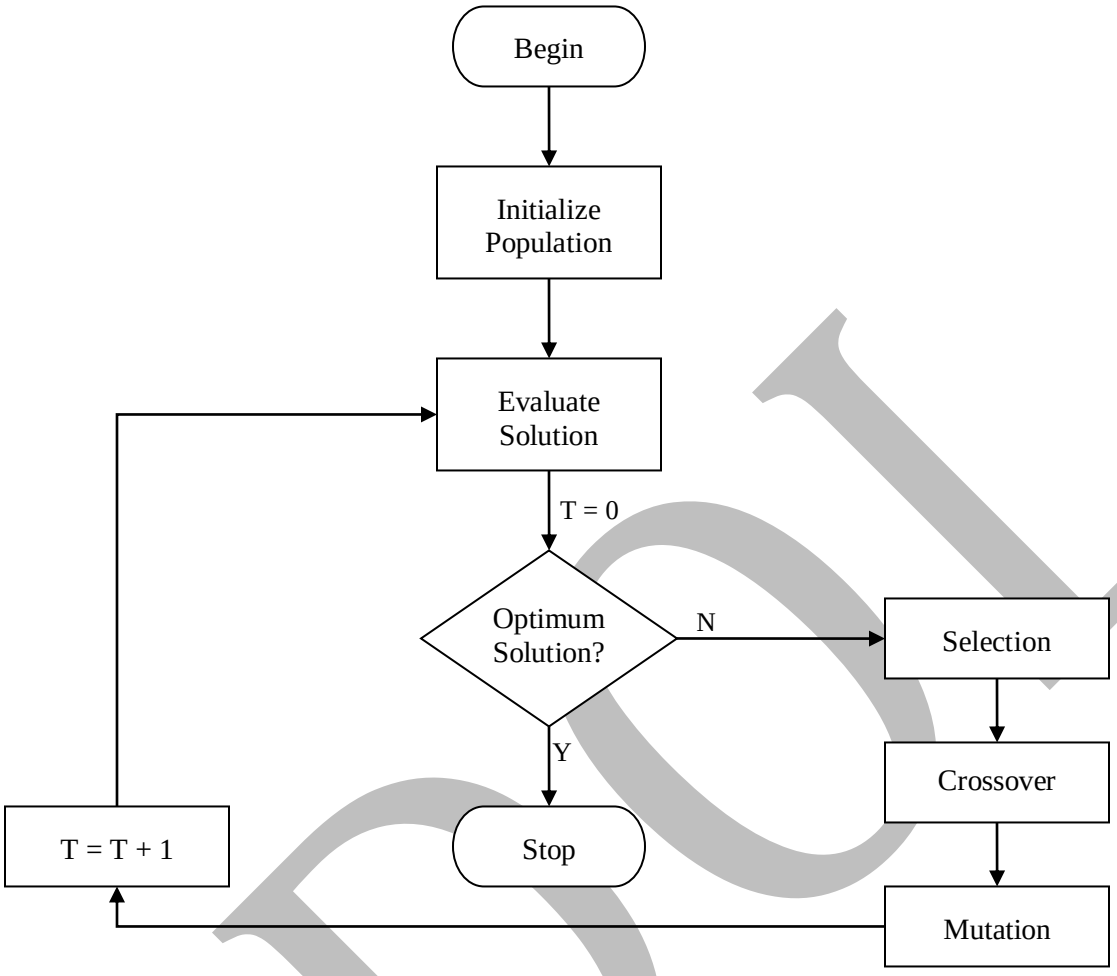
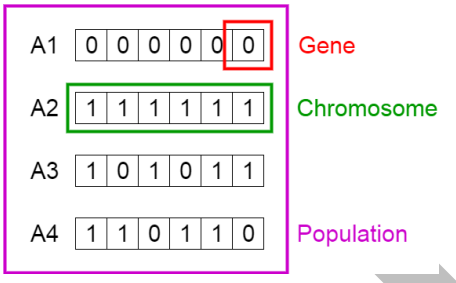


Figure 11.1: Flow of genetic algorithm

1. Initial Population

As described above, a gene is a string of bits. The *initial population* of genes (bitstrings) is usually created randomly. The length of the bitstring is depending on the problem to be solved.

First we create individuals and then we group them and call *Population.* An individual is distinguished by set of variables known as *Genes*. These Genes are combined into a string to form *Chromosome,* which is basically the solution.



2. Selection

As we now have the fitness score, in this phase we select the individuals with the highest scores and let them pass their genes to the next generations. After selecting individuals, we now group them into pair of two(parents) based on their fitness score because obviously it's not possible to reproduce without two entities. **Selection** means to extract a subset of genes from an existing (in the first step, from the initial -) population, according to any definition of *quality*. In fact, every gene must have a *meaning,* so one can derive any kind of a *quality measurement* from it - a "value". Following this quality "value" (*fitness*), Selection can be performed e.g. by Selection *proportional to fitness*:

i. Consider the population being rated, that means: *each gene has a related fitness*. (How to obtain this fitness will be explained in the Application-section.) The higher the value of the fitness, the better.
ii. The *mean-fitness* of the population will be calculated.
iii. Every individuum (=gene) will be copied as often to the new population, the better it fitness is, compared to the average fitness. *E.g.: the average fitness is 5.76, the fitness of one individuum is 20.21. This individuum will be copied 3 times.* All genes with a fitness at the average and below will be removed.
iv. Following this steps, one can prove, that in many cases the new population will be a little smaller, than the old one. So the new population will be filled up with randomly chosen individua from the old population to the size of the old one.

3. Crossover

The ultimate goal of reproduction is to mix the DNA of two individuals. So, we will do the same thing here. Let's take two individuals *Oswald* and *Daisy*, their DNA is defined by their alleles (value of each letter). Therefore, in order to mix their DNA, we just have to mix their letters.

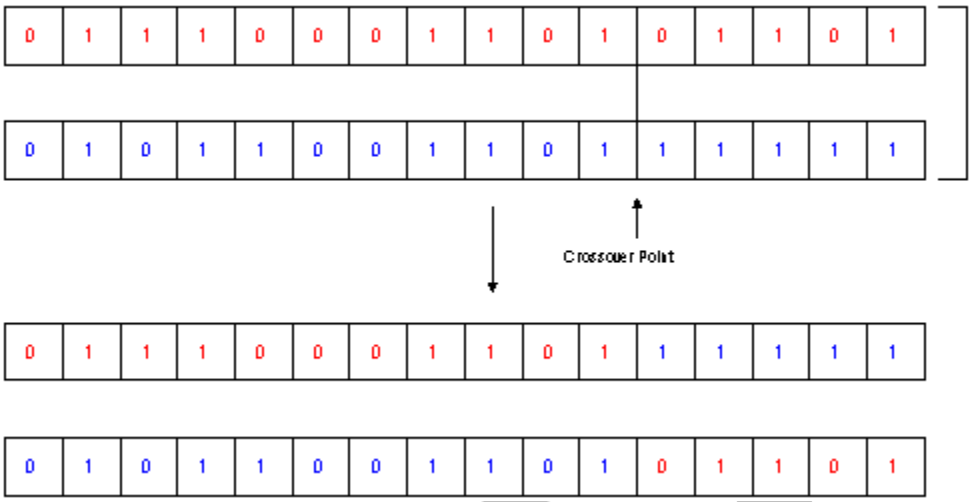


Figure 11.2: Crossover

The next steps in creating a new population are the **Mating** and **Crossover**: As described in the previous section there exist also a lot of different types of Mating/Crossover. One easy to understand type is the random mating with a defined probability and the *b_nX crossover* type. This type is described most often, as the parallel to the *Crossing Over* in genetics is evident:

a. P_M percent of the individual of the new population will be selected randomly and mated in pairs.
b. A *crossover point* (see fig.11.2) will be chosen for each pair
c. The information after the crossover-point will be exchanged between the two individual of each pair.

This crossover type usually offers higher performance in the search.

a. P_M percent of the individual of the new population will be selected randomly and mated in pairs.
b. With the probability P_C, two *bits* in the same position will be exchanged between the two individual. Thus not only **one***crossover point* is chosen, but *each bit* has a certain probability to get exchanged with its counterpart in the other gene.

4. Mutation

The primary goal of this step is to prevent our algorithm to be blocked in a local minimum. After the crossover, each individual must have a small probability to see change in their DNA.

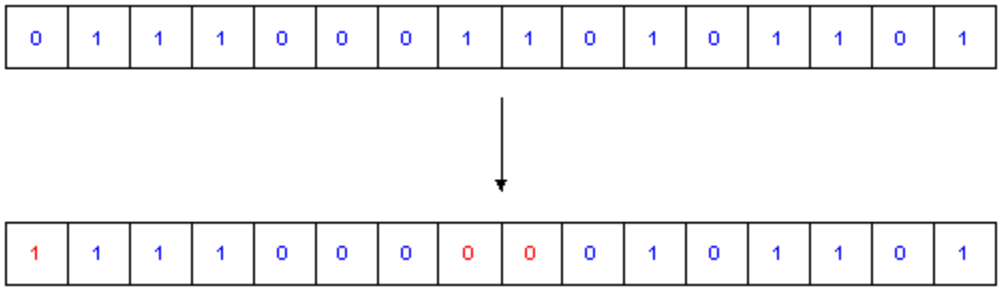


Figure 11.3: Mutation

The last step is the **Mutation**, with the sense of adding some effect of *exploration* of the phase-space to the algorithm. The implementation of Mutation is - compared to the other modules - fairly trivial: *Each bit* in *every gene* has a defined Probability *P* to get inverted.

The effect of *mutation* is in some way a antagonist to *selection*:

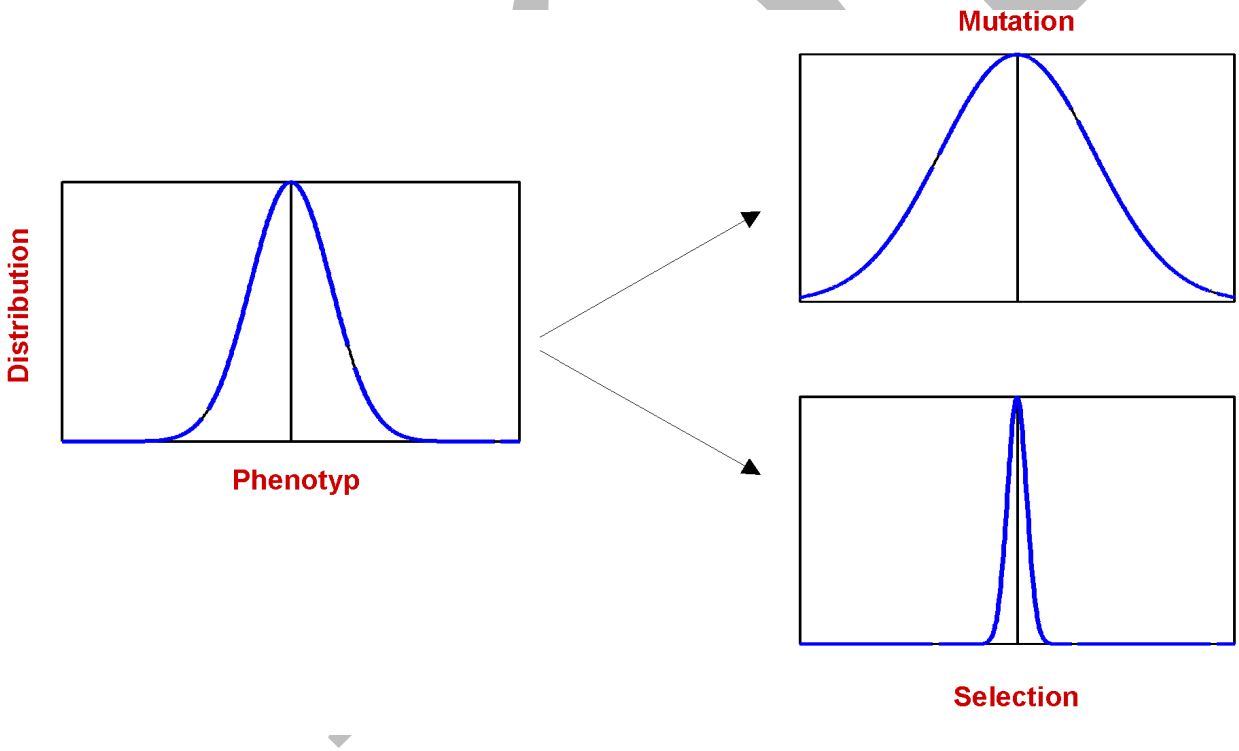


Figure 11.4: Influence of Selection and Mutation

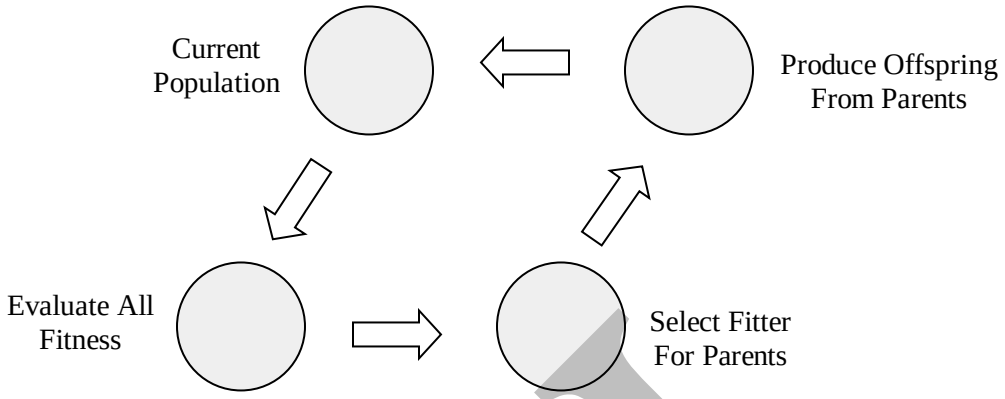


Figure 11.5: Genetic Algorithm

11.2.1 Simple Genetic Algorithm

```
Simple_Genetic_Algorithm()
{
        Initialize the Population;
        Calculate Fitness Function;

        While(Fitness Value != Optimal Value)
        {
                Selection; //Natural Selection, Survival Of Fittest
                Crossover; //Reproduction, Propagate favorable characteristics
                Mutation; //Mutation

                Calculate Fitness Function;
        }
}
```

11.3 The Genetic Algorithm Cycle of Reproduction

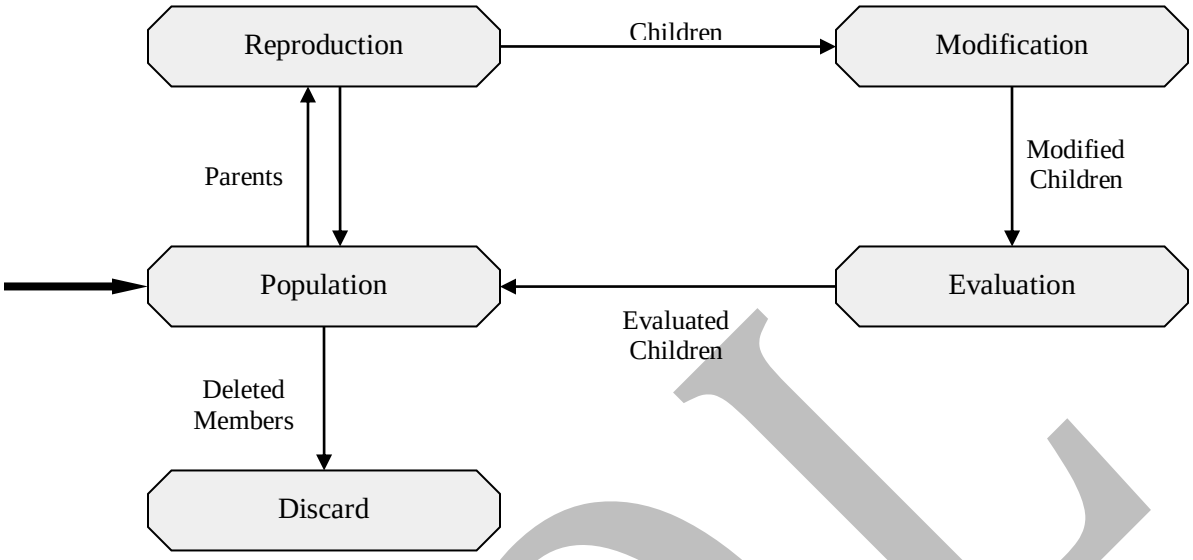


Figure 11.6: Reproduction

In the above figure 11.6 genetic reproduction cycle is explain with the help of algorithm. In the reproduction process selection of parent is an important part. Parents are selected at random with selection chances biased in relation to chromosome evaluations. Parent Selection is the process of selecting parents which mate and recombine to create off-springs for the next generation. Parent selection is very crucial to the convergence rate of the GA as good parents drive individuals to a better and fitter solutions.

However, care should be taken to prevent one extremely fit solution from taking over the entire population in a few generations, as this leads to the solutions being close to one another in the solution space thereby leading to a loss of diversity. **Maintaining good diversity** in the population is extremely crucial for the success of a GA. This taking up of the entire population by one extremely fit solution is known as **premature convergence** and is an undesirable condition in a GA.

11.3.1 Fitness Proportionate Selection

Fitness Proportionate Selection is one of the most popular ways of parent selection. In this every individual can become a parent with a probability which is proportional to its fitness. Therefore, fitter individuals have a higher chance of mating and propagating their features to the next generation. Therefore, such a selection strategy applies a selection pressure to the more fit individuals in the population, evolving better individuals over time.

Fitness Function

A fitness function quantifies the optimality of a solution (chromosome) so that that particular solution may be ranked against all the other solutions. A fitness value is assigned to each solution

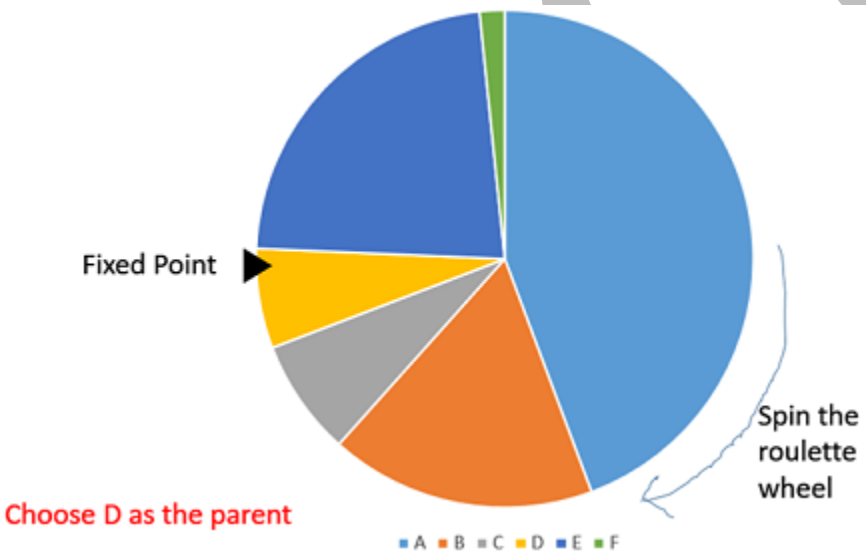
depending on how close it actually is to solving the problem. Ideal fitness function correlates closely to goal + quickly computable.Example. In TSP, f(x) is sum of distances between the cities in solution. The lesser the value, the fitter the solution is.

Consider a circular wheel. The wheel is divided into **n pies**, where n is the number of individuals in the population. Each individual gets a portion of the circle which is proportional to its fitness value.

Two implementations of fitness proportionate selection are possible −

Roulette Wheel Selection

In a roulette wheel selection, the circular wheel is divided as described before. A fixed point is chosen on the wheel circumference as shown and the wheel is rotated. The region of the wheel which comes in front of the fixed point is chosen as the parent. For the second parent, the same process is repeated.



Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

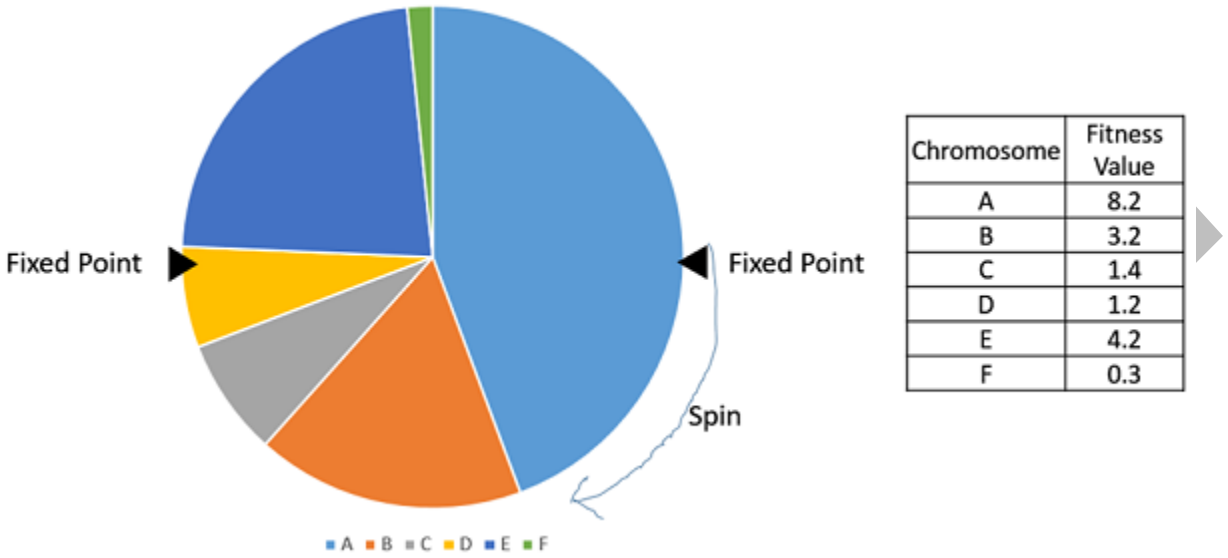
It is clear that a fitter individual has a greater pie on the wheel and therefore a greater chance of landing in front of the fixed point when the wheel is rotated. Therefore, the probability of choosing an individual depends directly on its fitness.

Implementation wise, we use the following steps −

- Calculate S = the sum of a finesses.
- Generate a random number between 0 and S.
- Starting from the top of the population, keep adding the finesses to the partial sum P, till P<S.
- The individual for which P exceeds S is the chosen individual.

Stochastic Universal Sampling (SUS)

Stochastic Universal Sampling is quite similar to Roulette wheel selection, however instead of having just one fixed point, we have multiple fixed points as shown in the following image. Therefore, all the parents are chosen in just one spin of the wheel. Also, such a setup encourages the highly fit individuals to be chosen at least once.

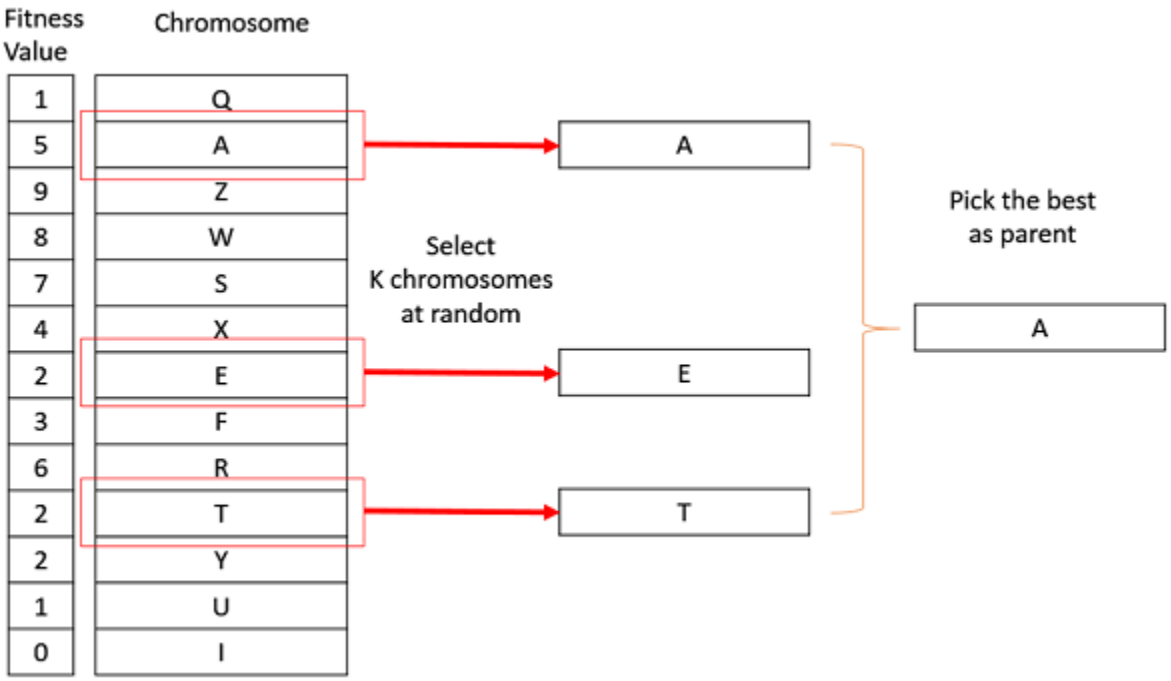


Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

It is to be noted that fitness proportionate selection methods don't work for cases where the fitness can take a negative value.

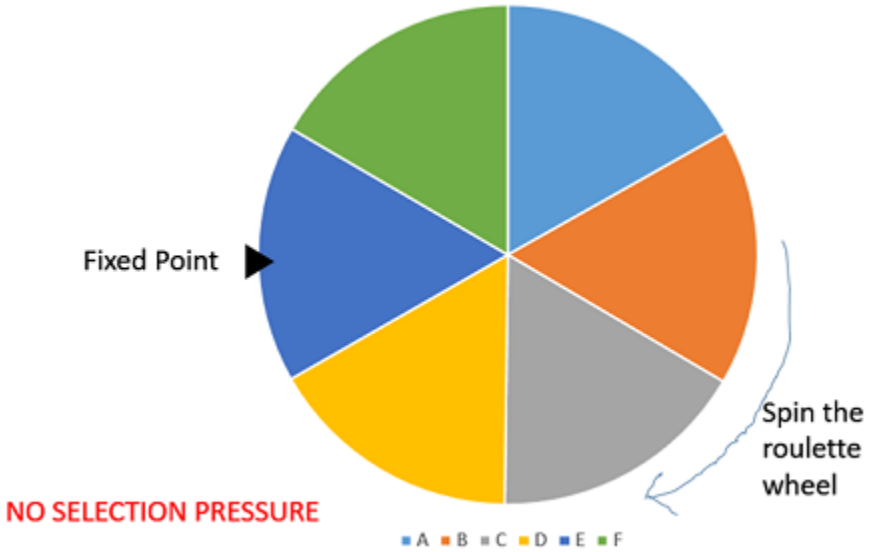
Tournament Selection

In K-Way tournament selection, we select K individuals from the population at random and select the best out of these to become a parent. The same process is repeated for selecting the next parent. Tournament Selection is also extremely popular in literature as it can even work with negative fitness values.



Rank Selection

Rank Selection also works with negative fitness values and is mostly used when the individuals in the population have very close fitness values (this happens usually at the end of the run). This leads to each individual having an almost equal share of the pie (like in case of fitness proportionate selection) as shown in the following image and hence each individual no matter how fit relative to each other has an approximately same probability of getting selected as a parent. This in turn leads to a loss in the selection pressure towards fitter individuals, making the GA to make poor parent selections in such situations.



Chromosome	Fitness Value
A	8.1
B	8.0
C	8.05
D	7.95
E	8.02
F	7.99

In this, we remove the concept of a fitness value while selecting a parent. However, every individual in the population is ranked according to their fitness. The selection of the parents depends on the rank of each individual and not the fitness. The higher ranked individuals are preferred more than the lower ranked ones.

Chromosome	Fitness Value	Rank
A	8.1	1
B	8.0	4
C	8.05	2
D	7.95	6
E	8.02	3
F	7.99	5

Random Selection

In this strategy we randomly select parents from the existing population. There is no selection pressure towards fitter individuals and therefore this strategy is usually avoided.

11.3.2 Crossover

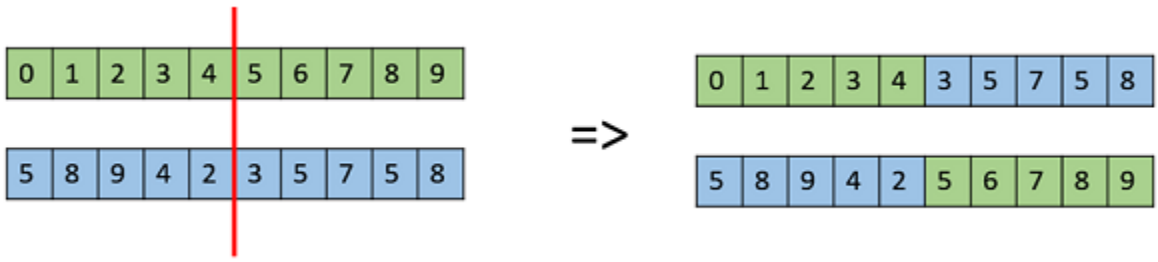
The crossover operator is analogous to reproduction and biological crossover. In this more than one parent is selected and one or more off-springs are produced using the genetic material of the parents. Crossover is usually applied in a GA with a high probability – p_c .

Crossover Operators

In this section we will discuss some of the most popularly used crossover operators. It is to be noted that these crossover operators are very generic and the GA Designer might choose to implement a problem-specific crossover operator as well.

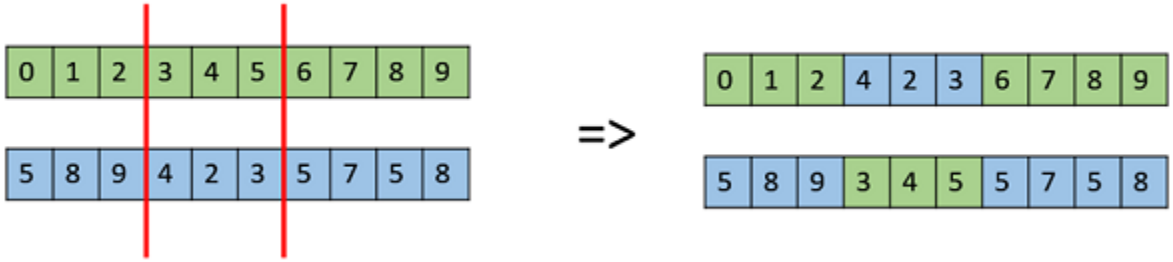
One Point Crossover

In this one-point crossover, a random crossover point is selected and the tails of its two parents are swapped to get new off-springs.



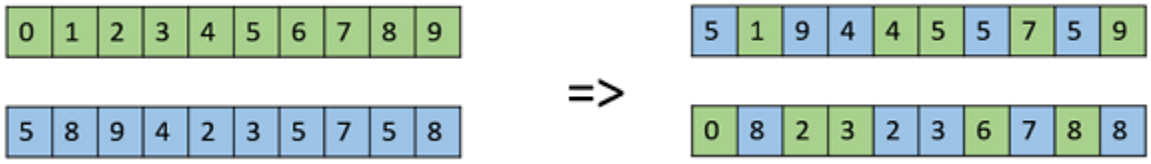
Multi Point Crossover

Multi point crossover is a generalization of the one-point crossover wherein alternating segments are swapped to get new off-springs.



Uniform Crossover

In a uniform crossover, we don't divide the chromosome into segments, rather we treat each gene separately. In this, we essentially flip a coin for each chromosome to decide whether or not it'll be included in the off-spring. We can also bias the coin to one parent, to have more genetic material in the child from that parent.

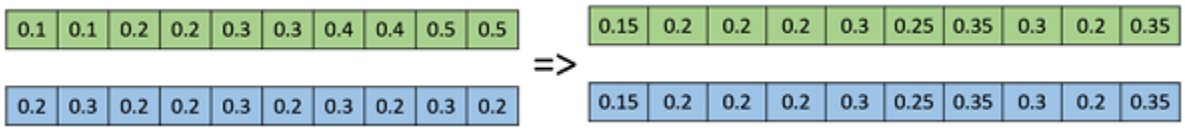


Whole Arithmetic Recombination

This is commonly used for integer representations and works by taking the weighted average of the two parents by using the following formulae −

- Child1 = α.x + (1-α).y
- Child2 = α.x + (1-α).y

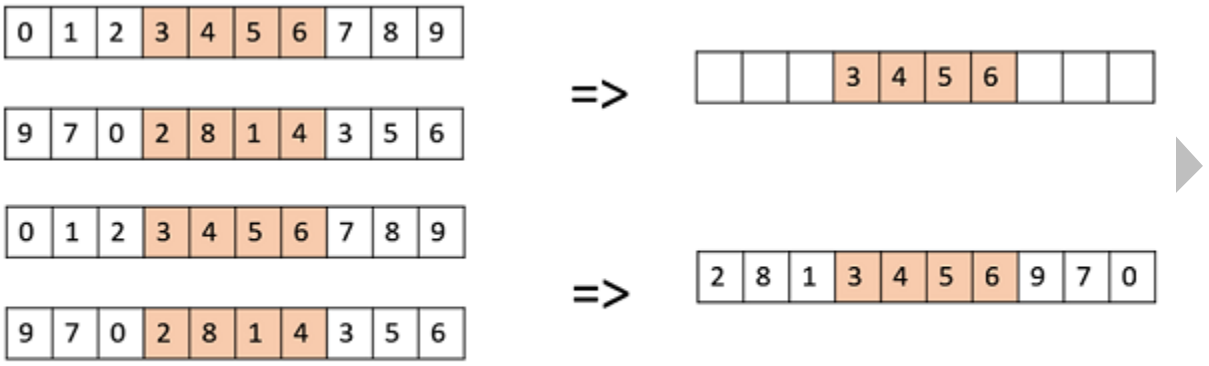
Obviously, if α = 0.5, then both the children will be identical as shown in the following image.



Davis' Order Crossover (OX1)

OX1 is used for permutation based crossovers with the intention of transmitting information about relative ordering to the off-springs. It works as follows −

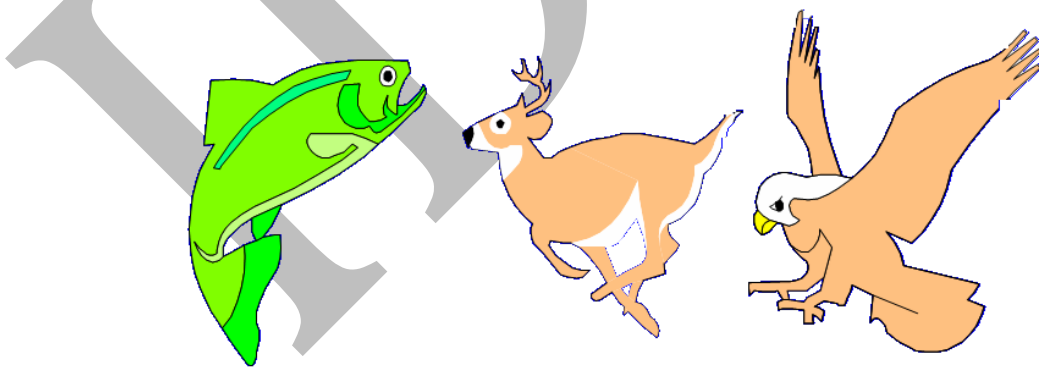
- Create two random crossover points in the parent and copy the segment between them from the first parent to the first offspring.
- Now, starting from the second crossover point in the second parent, copy the remaining unused numbers from the second parent to the first child, wrapping around the list.
- Repeat for the second child with the parent's role reversed.



Repeat the same procedure to get the second child

There exist a lot of other crossovers like Partially Mapped Crossover (PMX), Order based crossover (OX2), Shuffle Crossover, Ring Crossover, etc.

A Simple Example



- Encoding
  - The process of representing the solution in the form of a string that conveys the necessary information.
  - Just as in a chromosome, each gene controls a particular characteristic of the individual; similarly, each bit in the string represents a characteristic of the solution.
- Encoding Methods

- Binary Encoding
  - Most common method of encoding.
  - Chromosomes are strings of 1s and 0s and each position in the chromosome represents a particular characteristic of the problem.

Chromosome A	10110010110011100101
Chromosome B	11111110000000011111

- Permutation Encoding
  - Useful in ordering problems such as the Traveling Salesman Problem (TSP).
  - Example, in TSP, every chromosome is a string of numbers, each of which represents a city to be visited.

Chromosome A	1 5 3 2 6 4 7 9 8
Chromosome B	8 5 6 7 2 3 1 4 9

- Value Encoding
  - Used in problems where complicated values, such as real numbers, are used and where binary encoding would not suffice.
  - Good for some problems, but often necessary to develop some specific crossover and mutation techniques for these chromosomes.

Chromosome A	1.235 5.323 0.454 2.321 2.454
Chromosome B	(left), (back), (left), (right), (forward)

10.3.3 Mutation

It is the process by which a string is deliberately changed so as to maintain diversity in the population set.We saw in the giraffes' example, that mutations could be beneficial.Mutation may be defined as a small random tweak in the chromosome, to get a new solution. It is used to maintain and introduce diversity in the genetic population and is usually applied with a low probability – p_m. If the probability is very high, the GA gets reduced to a random search.

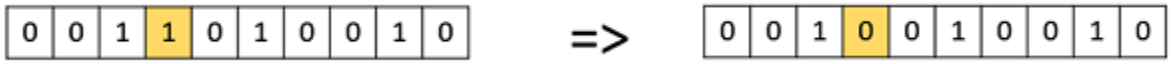
Mutation is the part of the GA which is related to the "exploration" of the search space. It has been observed that mutation is essential to the convergence of the GA while crossover is not.

Mutation Operators

In this section, we describe some of the most commonly used mutation operators. Like the crossover operators, this is not an exhaustive list and the GA designer might find a combination of these approaches or a problem-specific mutation operator more useful.

Bit Flip Mutation

In this bit flip mutation, we select one or more random bits and flip them. This is used for binary encoded GAs.

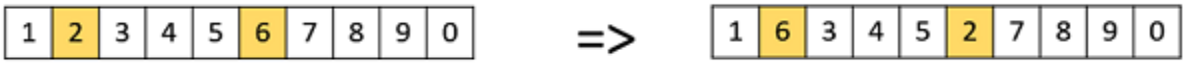


Random Resetting

Random Resetting is an extension of the bit flip for the integer representation. In this, a random value from the set of permissible values is assigned to a randomly chosen gene.

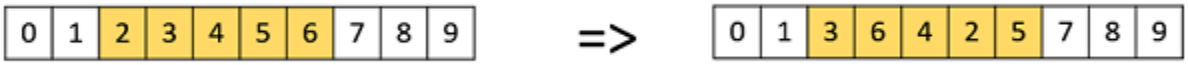
Swap Mutation

In swap mutation, we select two positions on the chromosome at random, and interchange the values. This is common in permutation based encodings.



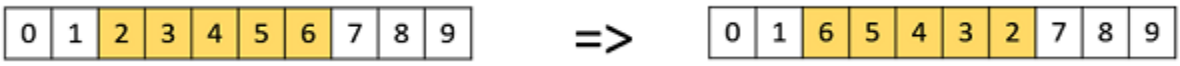
Scramble Mutation

Scramble mutation is also popular with permutation representations. In this, from the entire chromosome, a subset of genes is chosen and their values are scrambled or shuffled randomly.

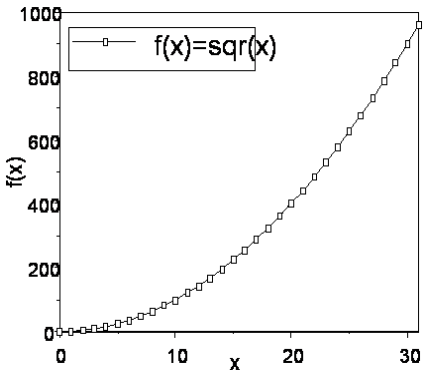


Inversion Mutation

In inversion mutation, we select a subset of genes like in scramble mutation, but instead of shuffling the subset, we merely invert the entire string in the subset.



A Simple Optimization Example



- Optimization of $f(x) = x^2$, with $x \varepsilon [0, 31]$
- Problem representation
    - Encoding of the variable x as a binary vector
    - [0, 31] ⌫ [00000, 11111]

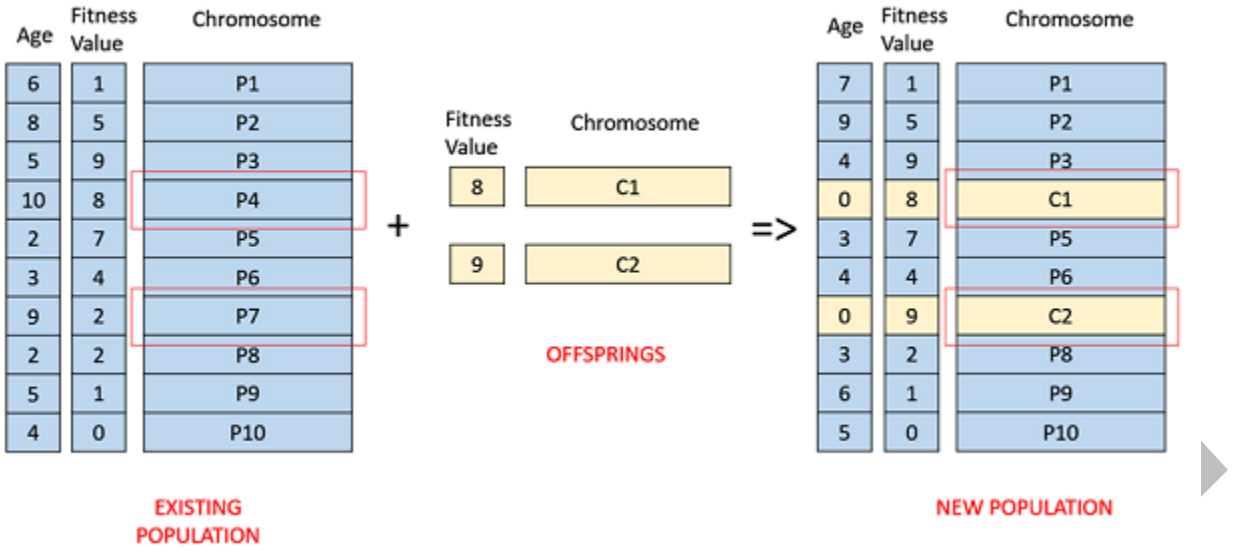
Some GAs employ **Elitism**. Elitism is a method which copies the best chromosome to the new offspring population before crossover and mutation. When creating a new population by crossover or mutation the best chromosome might be lost. Forces Genetic Algorithms to retain some number of the best individuals at each generation.Has been found that elitism significantly improves performance.

The easiest policy is to kick random members out of the population, but such an approach frequently has convergence issues, therefore the following strategies are widely used.

Age Based Selection

In Age-Based Selection, we don't have a notion of a fitness. It is based on the premise that each individual is allowed in the population for a finite generation where it is allowed to reproduce, after that, it is kicked out of the population no matter how good its fitness is.

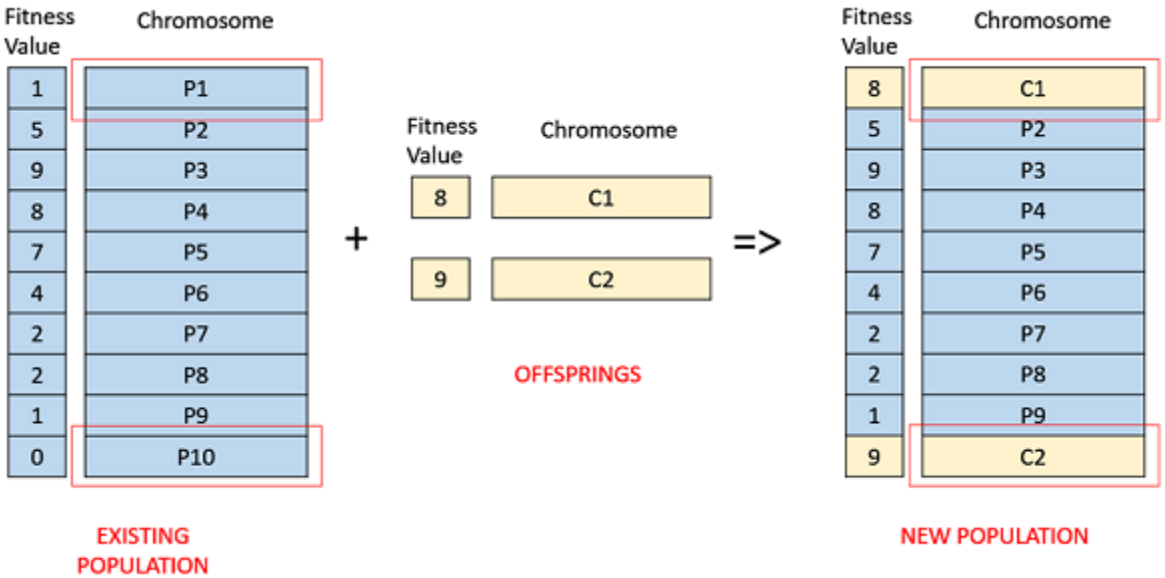
For instance, in the following example, the age is the number of generations for which the individual has been in the population. The oldest members of the population i.e. P4 and P7 are kicked out of the population and the ages of the rest of the members are incremented by one.



Fitness Based Selection

In this fitness based selection, the children tend to replace the least fit individuals in the population. The selection of the least fit individuals may be done using a variation of any of the selection policies described before – tournament selection, fitness proportionate selection, etc.

For example, in the following image, the children replace the least fit individuals P1 and P10 of the population. It is to be noted that since P1 and P9 have the same fitness value, the decision to remove which individual from the population is arbitrary.



The termination condition of a Genetic Algorithm is important in determining when a GA run will end. It has been observed that initially, the GA progresses very fast with better solutions coming in every few iterations, but this tends to saturate in the later stages where the improvements are very small. We usually want a termination condition such that our solution is close to the optimal, at the end of the run.

Usually, we keep one of the following termination conditions −

- When there has been no improvement in the population for X iterations.
- When we reach an absolute number of generations.
- When the objective function value has reached a certain pre-defined value.

For example, in a genetic algorithm we keep a counter which keeps track of the generations for which there has been no improvement in the population. Initially, we set this counter to zero. Each time we don't generate off-springs which are better than the individuals in the population, we increment the counter.

However, if the fitness any of the off-springs is better, then we reset the counter to zero. The algorithm terminates when the counter reaches a predetermined value.

Like other parameters of a GA, the termination condition is also highly problem specific and the GA designer should try out various options to see what suits his particular problem the best.

A GENETIC ALGORITHM with an Example

String No.	Initial Population	x Value	Fitness $f(x) = x^2$	% of Total Fitness	Selection Probability
1	01101	13	169	14.4	0.144
2	11000	24	576	49.2	0.492
3	01000	8	64	5.5	0.055
4	10011	19	361	30.9	0.309

After Selection	Mate	Crossover Point	Mutation	New Population	Fitness $f(x) = x^2$
0110 | 1	2	4	-	01100	144
1100 | 0	1	4	3	11101	841
11 | 000	4	2	-	11011	729
10 | 011	2	2	-	10000	256

11.4 Benefits of Genetic Algorithms

- Concept is easy to understand.

- Modular, separate from application.
- Supports multi-objective optimization.
- Good for "noisy" environments.
- Always an answer; answer gets better with time.
- Inherently parallel; easily distributed.
- Many ways to speed up and improve a Genetic Algorithm – based application as knowledge about problem domain is gained.
- Easy to exploit previous or alternate solutions.
- Flexible building blocks for hybrid applications.
- Substantial history and range of use.

11.4.1 When To Use A Genetic Algorithm?

- Alternate solutions are too slow or overly complicated.
- Need an exploratory tool to examine new approaches.
- Problem is similar to one that has already been successfully solved by using a Genetic Algorithm.
- Want to hybridize with an existing solution.
- Benefits of the Genetic Algorithm technology meet key problem requirements.

11.5 Some Genetic Algorithm Application Types

Domain	Application Types
Control	Gas Pipeline, Pole Balancing, Missile Evasion, Pursuit
Design	Semiconductor Layout, Aircraft Design, Keyboard Configuration, Communication Networks
Scheduling	Manufacturing, Facility Scheduling, Resource Allocation
Robotics	Trajectory Planning
Machine Learning	Designing Neural Networks, Improving Classification Algorithms, Classifier Systems
Signal Processing	Filter Design
Game Playing	Poker, Checkers, Prisoner's Dilemma
Combinatorial Optimization	Set Covering, Traveling Salesman, Routing, Bin Packing, Graph Coloring, Partitioning

Exercise

1. Give the outline of genetic algorithm.
2. Explain in brief the selection process along with its types.
3. Explain in brief about the cross process.
4. What is mutation? Explain with suitable example.
5. Explain in brief about elitism.
6. State the benefits of genetic algorithm.
7. Explain the applications of genetic algorithm.

Chapter 12

Mathematical Approach of Genetic Algorithm

Introduction:

Genetic Algorithms (GA) are alternative to the traditional optimization techniques by using directed random searches to locate optimal solutions.The GA search methods are rooted in the mechanisms of evolution and natural genetics. In nature, individuals best suited to competition for scanted resources survive. The various features that uniquely characterize an individual are determined by its genetic content.

Only the fittest individuals survive and reproduce, a natural phenomenon called "the survival of the fittest". The reproduction process generates diversity in the gene pool. Evolution is initiated when the genetic chromosomes from two parents recombine during reproduction. New combinations of genes are generated from previous ones and a new gene pool results. Specifically the exchange of genetic material among the chromosomes is called crossover. Segments of the two parent chromosomes are exchanged during crossover, creating the possibility of the right combination of genes for better individuals.The genetic algorithms manipulate a population of potential solutions to an optimization (or search) problem. They operate on encoded representations of the solutions, equivalent to the genetic material of individuals in the nature, and not directly on the solutions themselves. Usually the solutions are strings of bits from a binary alphabet.

As in nature, selection provides the necessary driving mechanism for better solutions to survive. Each solution is associated with a fitness value that reflects how good it is, compared with other solutions in the population. The higher the fitness value of an individual, the higher its chances for survival and reproduction.

Recombination of genetic material in the genetic algorithms is simulated through a crossover mechanism that exchanges portions between strings. Another operation, called mutation, causes sporadic and random alteration of the bits of strings. Mutation too has a direct analogy from nature and plays the role of regenerating lost genetic material.

Genetic Algorithm Structure

The genetic operators- *crossover* and *mutation*, generate, promote and juxtapose building blocks to form optimal strings. The *Crossover* tends to conserve the genetic information present in the strings to be crossed. *Mutation* generates radically new building blocks.

A Simple Genetic Algorithm

```
procedure simple-GA
begin
      t← 0
      Initialize P(T )
      Evaluate P(T )
      whileNOT( termination-condition ) do
          begin
              t←t + 1
              Select v, v1, v2 ∈P(T - 1 )
              Cross-over(v1, v2 )
              Mutate(v )
              Evaluate(v, v1, v2 )
          end
end.
```

Selection provides the favorable bias toward building blocks with higher fitness values and ensures that they increase in representation from generation to generation. The crucial operation is juxtaposing the building blocks achieved during crossover, and this is the cornerstone of the GA mechanics.

12.1Schema Theorem

Researchers have been trying to figure out the mathematics behind the working of genetic algorithms, and Holland's Schema Theorem is a step in that direction. Over the year's various improvements and suggestions have been done to the Schema Theorem to make it more general.

A genetic algorithm searches for the optimal string schemata in a competition with other ones from the current population in order to increase the number of the instances of the schemata in the next population. The notion that strings with high fitness values can be located by sampling schemata with high fitness values, called building blocks, and further combining these building blocks effectively is the building block hypothesis. The building block hypothesis assumes that the juxtaposition of good building blocks gives good strings.

When the effects of selection, crossover and mutation on the rate at which instances of schema increase from generation to generation are considered, one can see that proportionate selection increases or decreases the number in relation to the average fitness value of the schema. A schema must have a short defining rate too. Because crossover is disruptive, the higher the defining rate of a schema, the higher the probability that the crossover point will fall between its fixed positions and an instance will be destroyed. Thus schemata with high fitness values and small defining lengths grow exponentially with time. This is the essence of the schema theorem, which is the fundamental theorem of genetic algorithms. The fundamental schemata theorem of genetic algorithms states that schemata with high fitness values and small defining lengths grow exponentially in time. The following equation is a formal statement of the schemata theorem:

$$N(\mathbf{h}, t+1) >= N(\mathbf{h}, t) * f_i(\mathbf{h}, t) / f(t) [\, 1 - p_c * \delta(\mathbf{h}) / (l-1) - p_m * o(\mathbf{h}) \,]$$

where: $f_i(\mathbf{h}, t)$: average fitness value of schema **h** in generation t ;

$f(t)$: average fitness value of the population in generation t ;

p_c : crossover probability ;

p_m : mutation probability ;

$\delta(\mathbf{h})$: defining length of the schema **h** ;

$o(\mathbf{h})$: order of the schema **h** ;

$N(\mathbf{h}, t)$: expected number of instances of schema **h** in generation t ;

l : number of bit positions in a string.

In this section, we don't delve into the mathematics of the Schema Theorem, rather we try to develop a basic understanding of what the Schema Theorem is. The basic terminology to know are as follows −

- A **Schema** is a "template". Formally, it is a string over the alphabet = {0,1,*}, where * is don't care and can take any value.

  Therefore, *10*1 could mean 01001, 01011, 11001, or 11011

  Geometrically, a schema is a hyper-plane in the solution search space.

- **Order** of a schema is the number of specified fixed positions in a gene.

Schema	Order
***	0
101	3
*11	2
1**	1

- **Defining length** is the distance between the two furthest fixed symbols in the gene.

Schema	Defining Length
****	0
11	1
1*0*	2
1111	3

The schema theorem states that this schema with above average fitness, short defining length and lower order is more likely to survive crossover and mutation.

Consider 6-bit representations where * indicates don't care where,

0***** represents a subset of 32 strings , 1**00* represents a subset of 8 strings

Let H represent a schema such as 1**1**

Order: o(H)The number of fixed positions in the schema, H.

o(1*****) = 1,o(1**1*1) = 3

Length: delta(H)The distance between sentinel fixed positions in H.

delta(1**1**) = (4-1) = 3

delta(1*****) = 0

delta(***1**) = 0

In other words we can define the theorem as ,

Theorem: The number of representatives of any schema, S, increases in proportion to the observed relative performance of S.

12.1.1 Schemata: Properties

- All schemata are not equal.
- They differ in counts:
  - Order:
    - The order of a schema H, denoted by O(H), is the number of fixed positions present in the template of schema.
    - E.g. O(1*11*0*) = 4.
  - Defining length:
    - The defining length of schema H, denoted by δ(H), is the distance between the first and the last specific string position.
    - δ(1*11*0*) = 6 - 1.
    - δ(**1****) = 0.

12.2 Schema Difference Equation

Suppose at any given time t, there are m examples of a particular schema H in the population A(t).
It is denoted by m = m(H, t).
During reproduction, a particular string gets selected with probability,
$p_i = f_i / \Sigma f_i$, where f_i is the fitness.

Suppose a completely new generation is created from the population using reproduction.Then the number of schemata at time t is given:
$m(H, t+1) = m(H, t) * f(H) / f'$ where $f' = \Sigma f_i / n$.

Where, f(H) is the average fitness of strings representing schema H at t.And f' is the average fitness of the population.Thus, a schema grows as the ratio of the average fitness of the schema to the average fitness of the population.Schemata with fitness value above the population average will receive an increasing number of samples in the next generation.A schema grows or decays according to their schema averages under reproduction.If a particular schema H remains above an average amount cf', then

$m(H, t+1) = m(H, t) * (f' + cf') / f'$ i.e. $m(H, t+1) = (1+c) * m(H, t)$
When t = 0, $m(H, 1) = (1+c) * m(H, 0)$
When t = 1, $m(H, 2) = (1+c)^2 * m(H, 0)$ …
In general, $m(H, t) = (1+c)^t * m(H, 0)$

The equation is similar to compound interest, Geometric Progression i.e. reproduction allocations exponentially increasing or decreasing schemata to future generations.

12.2.1 Effect of Crossover on Schemata

Consider a string A = (0111100) and two schema H1 = (*1****0) and H2 = (***11**).Let the random crossover site be 3 i.e. A = (011|1100), H1 = (*1* | ***0) and H2 = (*** | 11**).Here schema H1 will be destroyed and H2 will be survived.Since the crossover site can uniformly between 1 and 6, as the defining length of H1 is large, H1 has lesser chance to survive.

Probability of H1 to be destroyed = 5 / 6.
In general, P(H1 to be destroyed) = $p_d = \delta(H1) / (l-1)$.
So, p(H1 to be survived) = $p_s = 1 - p_d$.
If p(crossover) = p_c, then $p_s = (1 - p_c) * \delta(H1) / (l-1)$.
So, combined effect of reproduction and crossover is
$m(H, t+1) >= m(H, t) * (f(H)/f') * ((1 - p_c) * \delta(H) / (l-1))$.

Thus, those schemata with above average fitness and short defining length will grow exponentially during evolution.

12.2.2 Effect of Mutation on Schema

Mutation is a permanent alteration with a probability p_m in the sequence that makes up a gene, such that the sequence differs from what is found in most people. Mutations range in size; they can affect anywhere from a single string building block (base pair) to a large segment of a chromosome that includes multiple genes. For a schema to be survived, each of the specified position should be survived of mutation.
So, survival probability for single position = $1 – p_m$

If O(H) is the fixed positions there in the string, p(survival of mutation) = $(1 – p_m) * O(H) = (1 - O(H)) * p_m + ... \Xi (1 - O(H)) * p_m$

Schema Processing: An Example

String No.	Initial Population	x Value	Fitness $f(x) = x^2$	% of Total Fitness	Selection Probability
1	01101	13	169	14.4	0.144
2	11000	24	576	49.2	0.492
3	01000	8	64	5.5	0.055
4	10011	19	361	30.9	0.309

After Selection	Mate	Crossover Point	Mutation	New Population	Fitness $f(x) = x^2$
0110 | 1	2	4	-	01100	144
1100 | 0	1	4	3	11101	841
11 | 000	4	2	-	11011	729
10 | 011	2	2	-	10000	256

Reproduction on H1

- Consider 3 schemata: H1 = 1****, H2 = *10** and H3 = 1***0.
- Strings 2 and 4 are representations of H1.
- So m(H1, t) = 2.
- After reproduction, there are 3 copies of H1.
- To check schema theorem,
- f(H1) = (576 + 361) / 2 = 468.5
- m(H1, t+1) = (f(H1) / f') * m(H1, t) = (468.5 / 293) * 2 = 3.20 = 3 (observed).

Crossover on H1

- No cross over since δ(H) = 0.

Mutation on H1

- If p_m = 0.001 then m * p_m = 3 * 0.001 = 0.003 = 0 i.e. no bits moved due to mutation in the schema.
- No mutation.
- Thus, we obtain the expected number of schemata as prescribed by the schema theorem.
- Similar is the case with H2 and H3.

12.3GA Based Machine Learning

Genetic Algorithms also find application in Machine Learning. **Classifier systems** are a form of **genetics-based machine learning** (GBML) system that are frequently used in the field of machine learning. GBML methods are a niche approach to machine learning.

There are two categories of GBML systems −

- **The Pittsburg Approach** − In this approach, one chromosome encoded one solution, and so fitness is assigned to solutions.
- **The Michigan Approach** − one solution is typically represented by many chromosomes and so fitness is assigned to partial solutions.

It should be kept in mind that the standard issue like crossover, mutation, Lamarckian or Darwinian, etc. are also present in the GBML systems.

12.3.1Two Armed And K – Armed Bandit Problem

K – Armed Bandit Problem is a problem in which a fixed limited set of resources must be allocated between competing (alternative) choices in a way that maximizes their expected gain, when each

choice's properties are only partially known at the time of allocation, and may become better understood as time passes or by allocating resources to the choice.This is a classic reinforcement learning problem that exemplifies the exploration-exploitation tradeoff dilemma. The name comes from imagining a gambler at a row of slot machines (sometimes known as "one-armed bandits"), who has to decide which machines to play, how many times to play each machine and in which order to play them, and whether to continue with the current machine or try a different machine. The multi-armed bandit problem also falls into the broad category of stochastic scheduling.

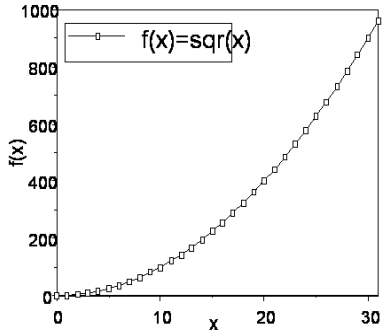
In the problem, each machine provides a random reward from a probability distribution specific to that machine. The objective of the gambler is to maximize the sum of rewards earned through a sequence of lever pulls. The crucial tradeoff the gambler faces at each trial is between "exploitation" of the machine that has the highest expected payoff and "exploration" to get more information about the expected payoffs of the other machines. The trade-off between exploration and exploitation is also faced in machine learning. In practice, multi-armed bandits have been used to model problems such as managing research projects in a large organization like a science foundation or a pharmaceutical company.

2 Armed Bandit Problem

- Slot machine with two arms: L and R.
- Each pays an award $\mu 1$or $\mu 2$ with variance $\sigma 12$ and $\sigma 22$, where $\mu 1 > \mu 2$.
- We want 2 things:
    - Make a decision about which arm to play.
    - Collect information about which is the better arm.
- First is called exploration and second is called exploitation.
- The trade-off between the exploration and exploitation of knowledge is a characteristic of adaptive systems.
- Experimentally, one can give exponentially increasing number of trials to the observed best of arms.
- This is similar to the exponential allocation to better schemata.

12.4 Building Block Hypothesis

Building Blocks are low order, low defining length schemata with the above given average fitness. The building block hypothesis says that such building blocks serve as a foundation for the GAs success and adaptation in GAs as it progresses by successively identifying and recombining such “building blocks”.



During crossover, these building blocks become exchanged and combined. So the Schema Theorem identifies the building blocks of a goodsolution although it only addresses the disruptive effects ofcrossover (and the constructive effects of crossover are supposed tobe a large part of why GA work).

Crossover combines short, low-order schemata into increasingly fit candidate solutions short low-order, high fitness schemata stepping stone solutions which combine Hi and Hj to create even higher fitness schemata. E.g. maximize function $f(x) = x^2$ on [0, 31].Let H1 = 1****

This 1-bit fixed schema corresponds to the right side of x = 16.

The 0-bit schema H2 = 0**** corresponds to the left side of x = 16.

The schema H3 = ****1 corresponds to the half domain between 1 & 2, between 3 &4, …

The schema H4 = ****0 corresponds to the half domain between 0 & 2, between 4 &6, …

Thus, 1-bit schemata contribute to the half domain of the full space.The schema H5 = 10*** corresponds to the domain between 16 & 24.The schema H6 = **1*1 contribute to the domain between 5 & 6, between 7 & 8, between 13 & 14 …

12.4.1 Genetic Algorithm Hard and Genetic Algorithm Deceptive

Schemata or building blocks lead to better population.But, not all problems can be solved using Genetic Algorithm way.The problems, which find difficult to solve using Genetic Algorithm techniques, are called Genetic Algorithm hard problems.Genetic Algorithm hard problems may have difficulties in coding.Possible solutions may not be amenable to genetic functions (operators).This coding-function combination of Genetic Algorithm hard problems is called Genetic Algorithm deceptive.

Example: an order-3 deception

"information represented by the schemata in the search space leads the search away from the global optimum, and instead directs the search toward the binary string that is the complement of the global optimum. The search space is order-3 deceptive. If the following relationships hold for the [three-bit]

schemata:"

0** > 1** and 00* > 11*, 01*, 10*

0 > *1* and 0*0 > 1*1, 0*1, 1*0

**0 > **1 and *00 > *11, *01, *10

but, 111 > 000, 001, 010, 100, 110, 101, 011

Example: f(000) = 28 f(100) = 14

f(001) = 26 f(101) = 10

f(010) = 22 f(110) = 5

f(011) = 20 f(111) = 30

Chaotic, noisy and "needle in a haystack" functionsGA-easy, GA-hard problems

12.4.2 Genetic Algorithm Deceptive: Characteristics

- Genetic Algorithm deceptive tends to have a remote, isolated optima i.e. a best point surrounded by a huge collection of worst points.
- Finding such is similar to finding a needle in a haystack.
- Many, not only Genetic Algorithm, techniques have difficulty in such cases.
- Consolation: such real world problems are less.

12.5 Minimum Deceptive Problem (MDP)

Is the smallest problem that can be deceptive or misleading?For this, we consider low order, short schema which lead to incorrect longer order schema.We can show that 2-bit schema problem is the smallest MDP.

2-BIT PROBLEM IS MDP

Consider four 2 order schema over two defining points with attached fitness.

0**0* → f_{00}
0**1* → f_{01}
1**0* → f_{10}
1**1* → f_{11}

(The fitness values are schema averages.)

Suppose f_{11} is the global maximum.Then $f_{11} > f_{00}, f_{01}, f_{10}$.Introduce an element deception to make Genetic Algorithm hard.For that, we assume that one or both of sub-optimal, 1 order schemata are better than the global optimal 1 order schemata

i.e. f(0*) > f(1*) and f(*0) > f(*1) i.e. (f(00) + f(01)) / 2 > (f(10) + f(11)) / 2 and (f(00) + f(10)) / 2 > (f(01) + f(11)) / 2.

Both cannot be true at the same time, as then f_{11} cannot be the global optimum.Only one result is true.

Without loss of generality, assume that f(0*) > f(1*) and f(*0) < f(*1).

Normalize the global conditions and label.

So $r = f_{11} / f_{00}$, $c = f_{01} / f_{00}$, $c' = f_{10} / f_{00}$ and r > c, r > 1 and r > c'

Deception condition in normalized form: r < 1 + c – c'

These results give: c' < 1 and c' < c

Thus there are two types of deceptive two-problems.

Type 1: $f_{01} > f_{00}$ (c > 1)

Type 2: $f_{00} > f_{01}$ (c <= 1)

Both are deceptive.
So 2-bit problem is deceptive.
Similarly we can prove that 1-bit problem is not deceptive.
So 2-bit problem is the MDP.

12.6 Extended Schema Analysis Of 2-Bit Problem

We have a 2-bit problem that seems misleading.By schema theorem,

$m(H, t+1) >= m(H, t) * (f(H)/f') * (1 – (p_c \delta(H)/(l-1)) – O(H) p_m)$

When $p_m = 0$, crossover has more importance.We look at a closer look at crossover.

Cross over yield table in 2-bit problem.

X	00	01	10	11
00	s	s	s	01, 10
01	s	s	00, 11	s
10	s	00, 11	s	s
11	01, 10	s	s	s

On crossover, complements lose genetic material.This loss is compensated by the gain to the other complementary pair schemata.We have to account for the expected loss and gain of schemata due to cross over.Assuming proportionate reproduction, crossover and mutation, we can have the proportion for each of the schema.

$P^{t+1}_{11} = P^t_{11} * f_{11} / f' \{1-p_c' (f_{00}/f') P^t_{11}\} + p_c' (f_{01}f_{10}/f'^2) P^t_{01} P^t_{10}$

$P^{t+1}_{10} = P^t_{10} * f_{10} / f' \{1-p_c' (f_{01}/f') P^t_{01}\} + p_c' (f_{00}f_{11}/f'^2) P^t_{00} P^t_{11}$

$P^{t+1}_{01} = P^t_{01} * f_{01} / f' \{1-p_c' (f_{10}/f') P^t_{10}\} + p_c' (f_{00}f_{11}/f'^2) P^t_{00} P^t_{11}$

$P^{t+1}_{00} = P^t_{00} * f_{00} / f' \{1-p_c' (f_{11}/f') P^t_{11}\} + p_c' (f_{01}f_{10}/f'^2) P^t_{01} P^t_{10}$

Where f' is average population fitness and $p_c' = p(\text{cross between bits}) = p_c * \delta(H) / (l-1)$.
These equations predict the expected proportions of the four schemata.
A necessary condition for Genetic Algorithm to be successful is that sequence $<P^t_{11}>$ converges to 1.
Thus, Genetic Algorithm refuses to be misled by initial conditions.

12.7 Fitness Scaling

- At start of Genetic Algorithm: some better fitter strings among inferiors.
- Within a few generations, the superior ones will become dominant.
- This may lead to premature convergence.
- Later though diverse, almost all may have almost same fitness.
- This may lead to average and superior ones get same copies in the next population.
- Survival of the fittest becomes a random walk, which needs to be avoided.
- Strategy to avoid scaling.

Linear Scaling

$f' = a f + b$ (a & b are to be found)

Always take:

- $f'_{avg} = f_{avg}$
- $f'_{min} = f_{min}$
- $2 * f'_{avg} = f_{max}$
- $f'_{max} = C_{mult} \cdot f_{avg}$

Where C_{mult} = expected number of copies of best in the next population.

Because Of Scaling

- Number of extraordinary best ones is restricted.
- Number of lowly ones increases.
- In mature run, a few bad strings may have below population average.
- If scaling is applied here, the low fitness values can go negative, which is undesired.
- In that case, take $f_{min} = f'_{min} = 0$.
- Scaling helps to prevent early dominance of a few best ones and encourages healthy competition among equals.

Coding In Genetic Algorithm

- There exist different methods. Under the different situations one the method is used from the given methods.
- Two guidelines
  - Principle of meaningful building blocks
    - The user should select a coding so that short order schemata are relevant to the underlying problem and relatively unrelated to schemata over other fixed positions.
  - Principle of minimal alphabets
    - The user should select the smallest alphabet that permits a natural expression of the problem.

Coding: Binary Vs. Non-Binary

- E.g. f(x) = maximize x^2 on [0, 31]

Binary Representation	32-Alphabet Representation
01101	N
11000	Y
01001	I
10011	T

- There are non-coding similarities to be exploited in non-binary.

Discretization of Continuum

- Many optimization problems require functions over a continuum.
- This can be converted into a finite collection of discrete problems which can then be solved using Genetic Algorithm.

Constraints

- Genetic Algorithm generally used for unconstrained problems.
- Genetic Algorithm can be used for constrained problems too.
- This can be done by incorporating a penalty in the objective function as and when the constraints are violated.

Exercise

1. Explain in brief about genetic algorithm.
2. Explain the stages of genetic algorithm.
3. Explain schemata theorem with suitable example.
4. Illustrate the schemata theorem.
5. What is Building Block?
6. Write a note on 2- Armed Bandit problem.
7. Write a note on k-armed bandit problem.
8. Explain in brief about minimum deceptive problem.
9. Explain genetic algorithm is hard and deceptive with suitable example.
10. Write a note on Extended Schema Analysis Of 2-Bit Problem.

Chapter 13

History and Applications of Genetic Algorithm

13.1 Introduction

"A genetic Algorithm is an iterative procedure maintaining a population of structures that are candidate solutions to specific domain challenges. During each temporal increment also called a generation, the structures in the current population are rated for their effectiveness as domain solutions, and on the basis of these evaluations, a new population of candidate solutions is formed using specific *genetic operators* such as reproduction, crossover, and mutation."

As define by Goldberg , "They combine survival of the fittest among string structures with a structured yet randomized information exchange to form a search algorithm with some of the innovative flair of human search. In every generation, a new set of artificial creatures (strings) is created using bits and pieces of the fittest of the old; an occasional new part is tried for good measure. Whilerandomized, genetic algorithms are no simple random walk. They efficiently exploit historical information to speculate on new search points with expected improved performance."

13.2 Genetic Algorithms Overview

It was introduced by developed by John Holland in 1975. Genetic Algorithms (GAs) are search algorithms based on the mechanics of the natural selection process (biological evolution). The most basic concept is that the strong tend to adapt and survive while the weak tend to die out. That is, optimization is based on evolution, and the "Survival of the fittest" concept. The genetic algorithm have the ability to create an initial population of feasible solutions, and then recombine them in a way to guide their search to only the most promising areas of the state space. Each feasible solution is encoded as a chromosome (string) also called a genotype, and eachchromosome is given a measure of fitness via a fitness (evaluation or objective) function. Thefitness of a chromosome determines its ability to survive and produce

offspring.

- A finite population of chromosomes is maintained.

GAs use probabilistic rules to evolve a population from one generation to the next. Thegenerations of the new solutions are developed by genetic recombination operators:

Biased Reproduction: selecting the fittest to reproduce:

- **Crossover**: combining parent chromosomes to produce children chromosomes
- **Mutation**: altering some genes in a chromosome.

Crossover combines the "fittest" chromosomes and passes superior genes to the next generation. Mutation ensures the entire state-space will be searched (given enough time) and can lead the population out of a local minima.

13.3 Uses of Genetic Algorithms

There are wide applications of genetic algorithms which can be employed in any of the concept. The following are some situations where genetic algorithms should be used:

- When an acceptable solution representation is available
- When a good fitness function is available
- When it is feasible to evaluate each potential solution
- When a near-optimal, but not optimal solution is acceptable.
- When the state-space is too large for other methods

13.4 Genetic Algorithms vs Traditional Algorithm

The traditional algorithms varies widely with genetic algorithms technically. Also the results obtained from genetic algorithms are much more effective.

1. As compare to traditional algorithms the genetic algorithms works with a coding of the parameters rather than the actual parameter.
2. The GA works from a population of strings instead of a single point.
3. Application of GA operators causes information from the previous generation to be carried

over

to the next.

4. The GA uses probabilistic transition rules, not deterministic rules.

13.5History of Genetic Algorithm

Genetic algorithms came from the research of John Holland, in the University of Michigan, in 1960. Their main purpose as to solve problems where deterministic algorithms are too costly. Genetic algorithms are part of the evolutionist algorithms category. These algorithms are "biosinspired" because they mimic living creatures' fitting in their environment for survival.

Genetic algorithms focus on the genetic material evolution inside a group of individuals. In each generation, individuals reproduce and share their genetic material. When applied to the population as a whole and followed on multiple generations, it is called genetic recombination.

Holland, the Father of Genetic AlgorithmIntroduced Genetic Algorithm as a computational technique, though was entirely different. He wanted to create general programs and machines which could adapt to the changing environment.Holland popularized the GA with the idea that in the future computers would be capable ofimplemented his more advanced methods.

Bagley and Adaptive Game Playing Program

- First coined Genetic Algorithm.
- He used to play hexa-pawn.
- He used reproduction, crossover, and mutation.
- He also reduced selection in the beginning to reduce dominance of some.
- Increased selection later to allow competition.
- The technique was called as ‘Adaptive Genetic Algorithm’.

Rosenberg and Biological Cell Simulation

- He simulated a population of a single celled organism.
- He also defined a finite length string with a pair of chromosome (diploid).
- He used a string length of 20 with a max of 16 alleles.
- And introduced ‘Offspring generation function (OGF)’ that fixes number of offspring to check selection.
- He also introduced p(crossover site).

Cavicchio and Pattern Recognition

- He applied Genetic Algorithm to the design of detectors for pattern recognition.
- An image is digitized as 25 x 25 binary pixel grid and a detector was used as a subset of the pixels.
- During training, known images are presented to generate the knowledge and a list of detector states are stored.
- During recognition phase, unknown image is presented and count the matching.
- He then allowed reproduction, crossover and mutation.
- He also used pre-selection – an offspring always replaces one of the parents to give diversity.

Weisenbergand Cell Simulation

- He developed computer simulation of a living cell.
- He also proposed a multilevel Genetic Algorithm.
- The lower one is an adaptive Genetic Algorithm and the upper is non-adaptive.
- Lower one is meant to find parameters of Genetic Algorithm.
- These parameters are given to the upper level Genetic Algorithm and test the fitness of population strings.
- The fitter ones will be sent to the lower level and continues.
- Upper one functions like a supreme judge.

Hollstein and Function Optimization

- He Used five selection methods listed as follows:
    - Progeny testing: fitness of offspring controls parent's further breeding.
    - Individual selection: fitness of one decides the future of it as a parent.
    - Family selection: fitness of family controls use of all family members as parent.
    - Within family selection: fitness of one within family controls selection within family.
    - Combined selection: combination of selection methods.
- He used eight mating preferences which includes:
    - Random mating: equally likely.
    - Inbreeding: related ones mate intentionally.
    - Line breeding: a unique one is identified and mate with standard one and offspring is selected.
    - Outbreeding: contrast ones are chosen as parents.
    - Self fertilization: breeds itself.
    - Clonal propagation: a copy of one is formed.
    - Positive assortive mating: like ones with like ones.
    - Negative assortive mating: unlike ones are bred.
- Used 16 string populations.
- And concluded that inbreeding and outbreeding are better.

Frantz And Positional Effect

- It includes larger population size of (100) and string size of (25).

- He used:
  - Roulette wheel selection
  - Simple crossover
  - Mutation
- The results illustrated a correlation between positional effect and rate improvement.
- He also introduced:
  - Inversion
  - Partial complement (migration)
  - Multiple point crossover
- He then use Migration:
  - To select a few strings.
  - To complement about one third of bits of these strings.
  - The new strings are called immigrants.
  - Helps in maintaining diversity, but reduces performance.

Bosworth, Foo, Zeigler – Real Genes

- The coding was based on minimalist – binary like - against maximalist approach.
- They thought mutation needed a change.
- So, they introduced five variations.
- The results showed that the technique does not sense as Genetic Algorithm.

Box and Evolution Operation

- This technique was more of a management technique for workers to execute a plan than an algorithm.
- Followed natures mechanism:
  - Genetic variability
  - Selection
- Loose application of mutation as anything which changes structure.
- Not a Genetic Algorithm in the modern sense.

Fogel, Queens And Wash – Evolution Programming

- They consider a state diagram of 3 state machines.
- 0 and 1 are the inputs.
- A, B, C are the states.
- α, β, γ are the outputs.
- Two operators:
  - Selection: choose best out of parent and child.
  - Mutation: make different a string by an output, state transition, no of states or initial state.
- Drawback: limited to small problem space.
- The transition description is given by:

Present State	Input Symbol	Next State	Output Symbol
C	0	B	β
B	1	C	α
C	1	A	γ
A	1	A	β
A	0	B	β
B	1	C	α

De-Jong & Function Optimization

- He mainly used as a function optimizer.
- He used six functions with the following properties:
  - Continuous / discreet
  - Convex / non-convex
  - Unimodal / multimodal
  - Quadratic / non-quadratic
  - Low dimensionality / high dimensionality
  - Deterministic / stochastic
- He devised two different performance measures:
  - Offline (convergence) performance
  - Online (ongoing) performance
- In offline: different functions are tried and the best is saved for subsequent operations.
- The performance xe(s) of strategy s on environment e is given by xe(s) = 1/T Σ fe(t) where fe(t) is the objective function value for environment e on trial t.
- In online: acceptable performance is taken.
- The online performance xe*(s) of strategy s on environment e is given by xe*(s) = 1/T Σ fe*(t) where fe*(t) = best{fe(1), fe(2), …, fe(t)}.
- De Jong called his algorithm reproduction plan R1.
- In R1, three operations were used:
  - Roulette wheel selection
  - Simple crossover
  - Simple mutation
- R1 is a family of plans using 4 parameters n, p_c, p_m and G (Generation Gap).
- G = 1 for non-overlapping populations
- = 0 < G < 1 for overlapping populations.
- In overlapping populations, n x G individuals will be selected for genetic operations.
- He observed that larger pop size lead to better offline performance and smaller pop size lead to rapid initial change.
- He investigated five variations of plan R1.
- They are:
  - R2 - Elitist Model
  - R3 - Expected Value Model
  - R4 - Elitist Expected Value Model
  - R5 - Crowding Factor Model
  - R6 - Generalized Crossover Model
- R2 - Elitist Model

- Let a*(t) be the best individual generated up to time t. After generating A(t+1) in the usual fashion, if a*(t) is not in A(t+1), then include a*(t) to A(t+1) as the $(N+1)^{th}$ member.
- It improves the local search at the expense of global perspective.

- R3 - Expected Value Model
  - Each string in the population is given an expected number of off-springs f / f'.
  - Thereafter, each time a string is selected for crossover or mutation, its offspring count is reduced by 0.5
  - When an individual is selected for reproduction without crossover or mutation, its offspring count is reduced 1.
  - If the offspring count < 0, it is no longer available for selection.
- R4 - Elitist Expected Value Model
  - Combination of R2 and R3.
  - Much better performance.
- R5 - Crowding Factor Model
  - In nature, like individuals dominate a niche in the population.
  - Then, increased competition for limited resources decreases life expectancy and birth rate.
  - De Jong enforced a crowding pressure by the forceful replacement of older strings with newer off-springs.
  - For that, consider an overlapping pop with G = 0.1
  - Defined a parameter-crowding factor.
  - When an off-spring is born, a string is selected for dying.
  - The dying string is selected as that one which resembles the new off-spring (like bit-by-bit similarity).
  - Process is similar to pre-selection of Cavicchio.
- R6 - Generalized Cross Over Model
  - Used a new parameter – number of crossover points (CP).
  - When CP = 1, it is simple crossover.
  - If l is the length of string, then there are ${}^{l}C_{CP}$ operators for multiple crossovers.
  - As CP is increased, each operator has less chance to be picked up during a particular cross and hence less structure can be preserved i.e. effectively, the process becomes a random shuffle and fewer important schemas can be preserved.

Improvement In Basic Techniques

- Since De Jong there were improvements to the basic Genetic Algorithm.
- They correspond to:
  - Selection
  - Scaling
  - Ranking

Alternative Selection Schema: Brindle

- Deterministic Sampling
  - Find $p_{selection} = f_i/\Sigma f_i$
  - E(Number of Strings) = e_i = int($p_{selection}$*n)
  - Population is selected according to the fraction part of e_i.

- Fill the remaining slots of population from the top of the sorted list.
- Example:

String No.	Initial Population	x Value	Fitness $f(x) = x^2$	% of Total Fitness	Selection Probability
1	01101	13	169	14.4	0.144
2	11000	24	576	49.2	0.492
3	01000	8	64	5.5	0.055
4	10011	19	361	30.9	0.309

- Here strings 2 and 4 are selected initially.
- Then sort the fractional parts 0.96, 0.56,0.23 and 0.22
- Best strings are 2 and 1 corresponding to the fractional parts 0.96 and 0.56
- Thus new population = {2,4,2,1}

- Remainder Stochastic Sampling Without Replacement
  - Like deterministic sampling, integer values are selected.
  - Fractional parts of e_i are taken as probabilities.
  - Bernoulli's trials are conducted with probability of success = fractional probabilities.
  - E.g. $e_i = 2.5$ will have 2 sure and another with probability 0.5
- Remainder Stochastic Sampling With Replacement
  - Like deterministic sampling, integer values are selected.
  - Fractional parts of e_i are used to calculate weights in a roulette wheel selection procedure.
- Stochastic Sampling With Replacement
  - Typical roulette selection.
- Stochastic Sampling Without Replacement
  - Typical De Jong's expected value modal R3.
  - Each string in the population is given an expected number of off-springs f / f`.
  - Thereafter, each time a string is selected for crossover or mutation, is offspring count is reduced by 0.5
  - When an individual is selected for reproduction without crossover or mutation, its offspring count is reduced 1.
  - If the offspring count < 0, it is no longer available for selection.
- Stochastic Tournament
  - Selection probabilities are calculated as normal.
  - Successive pairs of individuals are drawn using Roulette wheel selection.
  - Out of a pair, string with higher fitness is taken into the population.
  - A new pair is drawn and continued until the population is full.
- These selection procedures show many drawbacks.
- It is because of the inferiority of Roulette Wheel selection
- Out of all these, R3 is considered to be better.

13.6 Scaling Mechanism

At the beginning of the GA run, there may be a very high fitness individual i, that biases search towards *i*. Near the end of a run, when the population is converging, there may also not be much separation among individuals in the population. Neither is desirable. Thus we may want to scale the fitness so that selection pressure remains the same throughout the run. Let us formulate the problem in the following way:

- One useful scaling procedure is linear scaling where we want to scale the fitness of each individual in the population such that the scaled fitness is linearly related to the unscaled fitness.

$$f' = af + b$$

- We want to maintain a certain relationship between the maximum fitness individual in the population and the average population fitness. Let this be expressed by the following constraint equations:

$$f'_{max} = f_{avg} * C_s$$

$$f'_{avg} = f_{avg}$$

where *f'*is the scaled maximum fitness, $f_{avg}$is the average fitness of the population, and $C_5$is a scaling constant that specifies the expected number of copies of the best individual in the next generation. Increasing C_5will increase selection ``pressure" (bias towards best individual and quicker convergence), decreasing C_5will decrease selection pressure.

  - Sigma truncation
    - F' = f - (f^ - cσ) where σ = population standard deviation and c is a constant between 1 and 3.
    - Negative values will never occur.
  - Power low scaling
    - f' = f^k for some k.

Ranking Procedures

- Selection of strings is based on ranks of their fitness values.
- Population is sorted according to the fitness values.
- Individuals are assigned an offspring count based on their rank.

Applications Of Genetic Algorithm

- Medical image registration with Genetic Algorithm
  - Genetic Algorithm is used to perform image registration in a Digital Subtraction Angiography (DSA).

- DSA checks the interior of an artery by comparing two X-rays which are taken before and after injecting a dye.
- Images are digitized and subtracted pixel by pixel.
- The difference image gives the interior of the artery.
- The pre injection image is transformed by a bilinear map $x'(x, y) = a_0 + a_1x + a_2y + a_3y$ and $y'(x, y) = b_0 + b_1x + b_2y + b_3y$ where a_i and b_i are unknown.
- Genetic Algorithm is used to find this a_i and b_i by minimizing the mean absolute differences of the images.

- Iterated prisoner's Dilemma

		Player 2	
	Decision	Co-operate	Defect
Player 1	Co-operate	(R = 6, R = 6)	(S = 0, T = 10)
	Defect	(T = 10, S = 0)	(P = 2, P = 2)

- Play the problem repeatedly to find history of C and D.
- This iterative problem can be solved using 'tit for tat'.
- Axelord showed that this can be solved by a much better way using Genetic Algorithm by a representation of 63 bit string and using last three strategies of the other prisoner.

Good genetic algorithm: experiments show such as they:

- Choose high crossover probability.
- Choose low mutation probability.
- Choose moderate population size say 30.

13.7 Applications of Genetic Algorithms

Scheduling: Facility, Production, Job, and Transportation Scheduling

Design: Circuit board layout, Communication Network design,keyboard layout, Parametric design in aircraft

Control: Missile evasion, Gas pipeline control, Pole balancing

Machine Learning: Designing Neural Networks, Classifier Systems, Learning rules

Robotics: Trajectory Planning, Path planning

Combinatorial Optimization:TSP, Bin Packing, Set Covering, Graph Bisection, Routing.

Image Processing: Pattern recognition

Business: Economic Forecasting; Evaluating credit risks detecting stolen credit cards before

customer reports it is stolen

Medical: Studying health risks for a population exposed to toxins

Example Applications of Genetic AlgorithmsOrder-Based Genetic Algorithms

An order-based GA is where all chromosomes are permutation of the list. An order-based GAs greatly reduce the size of the search space by pruning solutions that we do not want to consider.Order-based GAs can be applied to a number of classic combinatorial optimization problems such as: TSP, Bin Packing, Package Placement, Job Scheduling, Network Routing, Vehicle Routing, various layout problems, etc.Crossover functions for order-based GAs include Edge Recombination, Order Crossover #1, Order Crossover #2, PMX, Cycle Crossover, Position Crossover, etc. Whitley and Starkweather.PMX (Partially Matched Crossover)

```
   Parent 1   3 7 1 9 | 6 4 5 | 2 8
  Parent 2   4 7 8 5 | 3 9 2 | 1 6
 ( 6<--> 3 )  ( 4<--> 9 )   ( 5<--> 2 )
   Child 1    6 7 1 4 | 3 9 2 | 5 8
   Child 2    9 7 8 2 | 6 4 5 | 1 3
```

Mutation functions for order-based GAs include

Swap two elements

```
   *         *                 *         *
9 8 7 6 5 4 3 2 1    ==>    9 3 7 6 5 4 8 2 1
```

Move one element

```
     *                                   *
9 8 7 6 5 4 3 2 1    ==>    9 8 6 5 4 3 7 2 1
```

Reorder a sublist

```
9 8 | 7 6 5 4 3 | 2 1  ==>  9 8 | 5 3 4 6 7 | 2 1
```

Traveling Salesman Problem

Example TSP GA executions adjusting pool size:

```
                    TSP 1024 Cities.
PoolSize       500             250             125
```

Length_String 1024

Trials 100000

Bias 1.9

RandomSeed 15394157 <same><same>

MutateRate 0.15

NodeFile cities1024

StatusInterval 5000

RESULT = 116987 88436 90906

TSP 320 Cities.

PoolSize 2000 1000 500 250 125

Length_String 320

Trials 100000[a]

Bias 1.9

RandomSeed 15394157 <same><same><same><same>

MutateRate 0.15

NodeFile cities320

StatusInterval 1000

RESULTS:

Best:	30,761[b]	25,708	21,366	18,676[c]	23,760[d]
Worst:	35,102	28,366	23,235	18,676	23,760
Average:	34,209	27,863	22,880	18,676	23,760

aApoolsize of 2000 for 205,000 trials yielded best of 22,777

b CPU time on a SPARC 1+ was approximately 100 minutes.

c Converged after 72,000 trials

d Converged prematurely after 33,000 trials

TSP 105 Cities.

PoolSize	750	500	250	125
Length_String	105			
Trials	70000			
Bias	1.9			
RandomSeed	15394157	<same>	<same>	<same>
MutateRate	0.15			
NodeFile	cities105			
StatusInterval	1000			
RESULTS:				
Trials at				
Convergence:	109,000	61,000	32,000	9,000
Best:	16,503	17,193	24,079	32,370
Worst:	16,503	17,193	24,079	32,370
Average:	16,503	17,193	24,079	32,370

13.8 Classifier Systems

- According to Goldberg, a classifier system is "a machine learning system that learns syntactically simple string rules to guide its performance in an arbitrary environment".
- A classifier system consists of three main components:
  - Rule and message system
  - Apportionment of credit system
  - Genetic Algorithm (for evolving classifiers)
- First implemented in a system called CS1 by Holland/Reitman(1978).
- Example of classifer rules:

  00##:0000
  00#0:1100
  11##:1000
  ##00:0001
- Fitness of a classifier is defined by its surrounding environments that pays payoff to classifiers and extract fees from classifiers.
- Classifier systems employ a Michigan approach(populations consist of single rules) in the context of an externally defined fitness function.

13.8.1 Rule & Message System

- It is a kind of production system.
- The production rules have syntax as follows:
  - If <condition> then <action>

- E.g. if <(0,0)> then <(4,0)>

- Each rule is of a fixed length and is amenable to genetic operations.
- It allows parallel action of rules a single of usual expert system.
- The relative value of a rule is to be learned against fixed value.
- Competition ensures good rules survive.
- Classifier's bank balance is taken as fitness and an internal currency is introduced.
- <message> :: $\{0, 1\}^l$ is the basic token of info exchange.
- <condition > :: $\{0, 1, \#\}^l$
- A condition is matched by a message if the corresponding bits are mapped.
- E.g. #10# matches 1100 but not 1000.
- Once matched, that classifier becomes candidate to post its message to the message list.
- Example:

No.	Classifier
1	01## : 0000
2	00#0 : 1100
3	11## : 1000
4	##00 : 0001

Suppose environment sends message 0111.

13.9 Apportionment of Credit Algorithm

Credit Apportionment scheme is the backbone of the performance of adaptive rule based system. The more cases the credit apportionment scheme can consider, the better is the overall systems performance.

- Bucket Brigade Algorithm: rank the individual classifier according to its efficiency in achieving reward from the environment.
- It is an 'info economy', where right to trade is bought & sold by classifiers.
- They form a chain of middlemen from the info manufactures (environment) to the info customers (effectors).
- Has two components:
  - Auction
    - When a classifier matches a condition, then it is qualified for auction.
    - Each classifier maintains a record of its net worth, called strength.
    - Each matched classifier makes a Bid B proportional to its strength.
    - Thus, highly fit ones are given preference.
  - Clearing House
    - When a classifier is selected for auction, it clears payment through clearing house.
    - A matched & activated classifier sends its bids to those classifiers responsible for sending the messages that matched bidding classifier's conditions.
- Classifiers make bids (B_i) during the auction.
- Winning classifiers turn over their bids to the clearing house as payments (P_i).

- A classifier may have receipts R_i from its previous message sending activity or from environment.
- Thus, the strength of i^{th} classifier $S_i(t+1) = S_i(t) – P_i(t) – T_i(t) + R_i(t)$.
- But, bids are corresponding to its strengths, so $B_i = C_{bid} \times S_i$.
- If there are noise, take the effective bid as $E_{B_i} = B_i + N(\sigma \text{ bid})$.
- The winner pay their bids B_i not E_{B_i} to the clearing house.
- The tax of the classifier is given by $T_i = C_{tax} \times S_i$.
- So the difference equation is given by $S(t+1) = S(t) – C_{bid} S(t) – C_{tax} S(t) + R(t)$
- I.e. $S(t+1) = (1 - k) S(t) + R(t)$ where $k = C_{bid} + C_{tax}$.
- The system is stable only when R(t) is bounded.
- Leaving R(t), $S(t+1) = (1 – k) S(t)$
- … $S(n) = (1 – k)^n S(0)$, for a active classifier.In order to apply new and better rules into the system of bucket brigade, Genetic Algorithm is used.Here it is different from the optimization case.Non overlapping population is possible.Generation gap is used.
- Selection is through RW and De Jong's crowding procedure.
- Mutation with change as 0 → {1, #} with probability.

13.9.1 Simple Classifier System (SCS)

- A SCS, a simple version of classifier system is developed.
- Experiment results show SCS with Genetic Algorithm outperform SCS without Genetic Algorithm and random guessing.
- GBML systems discuss better computer programs by applying selection, recombination and other genetic operators.
- Holland pioneered theoretical foundation of GBML.
- This followed:
    - Proposal of Broadcast Language.
    - Implementation of first classifier system (Cognitive System – 1)
- Proposal to construct complex machines built from fixed components with schemata property.

CS-1

- Classifier conditions are constructed over {0, 1, #}.
- Many resources - e.g. hunger, thirst.
- Maintains different reservoirs for different resources.
- Current resource levels are used to determine current demand, which then are used to determine which rules to activate.
- An epochal algorithm is used instead of bucket brigade algorithm.
- Epoch is the time period between two payoff events.
- The parameters are the predicted payoff values u_i.
- Let d_i be the current demand (i.e. lower reservoir level).
- Then appropriation value $\alpha = \Sigma d_i u_i$.
- Roulette wheel is weighted with αM where M = match score, which increases with rule specificity.
- Roulette wheel selection is used to select winner.

- The epochal apportionment algorithm tracks the accuracy of a classifier's predicted payoff using three parameters:
    - Age
    - Frequency
    - Attenuation
- Results show the procedure outperforms other methods.

LS-1 SYSTEM

- Smith's LS-1 system has different architecture.
- Holland's CS treat rules as individuals whereas LS-1 treats rule sets as individuals in a population.
- Has four genetic operators:
    - Reproduction
    - Mutation
    - Modified Crossover
    - Inversion (i.e. r1:r2:r3 → r3:r2:r1).
- Results: good, but cannot be compared to CS-1 as measures are different.

13.9.1 Eye–Eye Coordination

- Wilson's classifier system for the sensory-motor coordination of a video camera.
- To learn to center an object in the video camera by moving the camera in the right direction.
- Uses an innovated CS-1 architecture.
- Uses a complex retina-cortex mapping.
- Instead of 1-dimension string, 4 by 4 arrays are used as rules.
- Uses 2-dimension crossover.

13.9.2 Animat Classifier System

- Wilson's roaming classifier system that searches two dimensional woods, seeking food and avoiding trees.
- Uses a 18 by 58 rectangular grid which contains trees(T) and food(F).
- The ANIMAT, represented by *, has the knowledge about immediate surroundings.
- E.g.
    - BTT
    - B*F
    - BBB
- It generates an environmental message TTFBBBBB.
- Take T – 01, F-11 and B – 00.
- Then 0101110000000000 is the bit representation.
- There are eight classifiers to recognize this.
- There are four genetic operators:
    - Match set, set of matching classifiers.
    - Create op, when no matching classifier.

- Partial intersection op: 2 rules with same action are aligned by replacing mismatch with #.
- Time to payoff estimation.

Exercise:

1. Explain the evolution of genetic algorithm.
2. Give comparison between genetic and traditional algorithm.
3. Explain the applications of genetic algorithms.
4. Explain in brief about classifier system.
5. Explain De-Jong & Function Optimization with example.
6. Explain Apportionment of Credit Algorithm.
7. Explain Animat Classifier System with suitable.

Chapter 14
Introduction to Data Warehousing

14.1 Introduction

The term "Data Warehouse" was first coined by Bill Inmon in 1990. According to Inmon, a data warehouse is a subject oriented, integrated, time-variant, and non-volatile collection of data. This data helps analysts to take informed decisions in an organization.An operational database undergoes frequent changes on a daily basis on account of the transactions that take place. Suppose a business executive wants to analyze previous feedback on any data such as a product, a supplier, or any consumer data, then the executive will have no data available to analyze because the previous data has been updated due to transactions.

A data warehouses provides us generalized and consolidated data in multidimensional view. Along with generalized and consolidated view of data, a data warehouses also provides us Online Analytical Processing (OLAP) tools. These tools help us in interactive and effective analysis of data in a multidimensional space. This analysis results in data generalization and data mining.Data mining functions such as association, clustering, classification, prediction can be integrated with OLAP operations to enhance the interactive mining of knowledge at multiple level of abstraction. That's why data warehouse has now become an important platform for data analysis and online analytical processing.

14.2 Datawarehousing

A data warehousing is a technique for collecting and managing data from varied sources to provide meaningful business insights. It is a blend of technologies and components which allows the strategic use of data.

It is electronic storage of a large amount of information by a business which is designed for query and analysis instead of transaction processing. It is a process of transforming data into information and making it available to users in a timely manner to make a difference.

The decision support database (Data Warehouse) is maintained separately from the organization's operational database. However, the data warehouse is not a product but an environment. It is an architectural construct of an information system which provides users with current and historical decision support information which is difficult to access or present in the traditional operational data store

Data warehouse system is also aligned with:

- Decision Support System (DSS)
- Executive Information System
- Management Information System
- Business Intelligence Solution
- Analytic Application
- Data Warehouse

Figure 14.1: DatawarehouseApproaches

14.3 History of Datawarehouse

The Datawarehouse benefits users to understand and enhance their organization's performance. The need to warehouse data evolved as computer systems became more complex and needed to handle increasing amounts of Information. However, Data Warehousing is a not a new thing.

Here are some key events in evolution of Data Warehouse-

- 1960- Dartmouth and General Mills in a joint research project, develop the terms dimensions and facts.
- 1970- A Nielsen and IRI introduces dimensional data marts for retail sales.
- 1983- Tera Data Corporation introduces a database management system which is specifically designed for decision support
- Data warehousing started in the late 1980s when IBM worker Paul Murphy and Barry Devlin developed the Business Data Warehouse.
- However, the real concept was given by Inmon Bill. He was considered as a father of data warehouse. He had written about a variety of topics for building, usage, and maintenance of the warehouse & the Corporate Information Factory.
- Challenge: How to manage ever-increasing amounts of information.
- Solution: Data Mining and Knowledge Discovery Databases (KDD).

14.4 Comparison between Data Warehouse and Operational Databases

A data warehouses is kept separate from operational databases due to the following reasons −

- An operational database is constructed for well-known tasks and workloads such as searching particular records, indexing, etc. In contract, data warehouse queries are often complex and they present a general form of data.
- Operational databases support concurrent processing of multiple transactions. Concurrency control and recovery mechanisms are required for operational databases to ensure robustness and consistency of the database.
- An operational database query allows to read and modify operations, while an OLAP query needs only **read only** access of stored data.
- An operational database maintains current data. On the other hand, a data warehouse maintains historical data.

14.5 Data Warehouse Features

The key features of a data warehouse are discussed below −

- **Subject Oriented** − A data warehouse is subject oriented because it provides information around a subject rather than the organization's ongoing operations. These subjects can be product, customers, suppliers, sales, revenue, etc. A data warehouse does not focus on the ongoing operations, rather it focuses on modelling and analysis of data for decision making.
- **Integrated** − A data warehouse is constructed by integrating data from heterogeneous sources such as relational databases, flat files, etc. This integration enhances the effective analysis of data.
- **Time Variant** − The data collected in a data warehouse is identified with a particular time period. The data in a data warehouse provides information from the historical point of view.
- **Non-volatile** − Non-volatile means the previous data is not erased when new data is added to it. A data warehouse is kept separate from the operational database and therefore frequent changes in operational database is not reflected in the data warehouse.

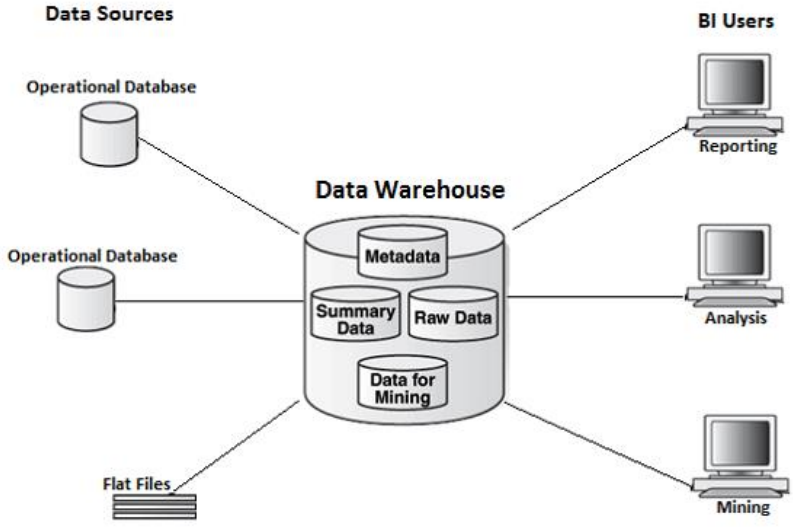


Figure 14.2: DatawarehouseSystem

Components of Data warehouse

Four components of Data Warehouses are:

Load manager: Load manager is also called the front component. It performs with all the operations associated with the extraction and load of data into the warehouse. These operations include transformations to prepare the data for entering into the Data warehouse.

Warehouse Manager: Warehouse manager performs operations associated with the management of the data in the warehouse. It performs operations like analysis of data to ensure consistency, creation of indexes and views, generation of denormalization and aggregations, transformation and merging of source data and archiving and baking-up data.

Query Manager: Query manageris also known as backend component. It performs all the operation operations related to the management of user queries. The operations of this Data warehouse components are direct queries to the appropriate tables for scheduling the execution of queries.

End-user access tools:

This is categorized into five different groups like 1. Data Reporting 2. Query Tools 3. Application development tools 4. EIS tools, 5. OLAP tools and data mining tools.

14.5.1 Data Warehouse Applications

As discussed before, a data warehouse helps business executives to organize, analyze, and use their data for decision making. A data warehouse serves as a sole part of a plan-execute-assess "closed-loop" feedback system for the enterprise management. Data warehouses are widely used in the following fields −

- Financial services
- Banking services
- Consumer goods
- Retail sectors
- Controlled manufacturing

14.6 Types of Data Warehouse

Information processing, analytical processing, and data mining are the three types of data warehouseapplications that are discussed below −

- **Information Processing** − A data warehouse allows to process the data stored in it. The data can be processed by means of querying, basic statistical analysis, reporting using crosstabs, tables, charts, or graphs.

- **Analytical Processing** − A data warehouse supports analytical processing of the information stored in it. The data can be analyzed by means of basic OLAP operations, including slice-and-dice, drill down, drill up, and pivoting.
- **Data Mining** − Data mining supports knowledge discovery by finding hidden patterns and associations, constructing analytical models, performing classification and prediction. These mining results can be presented using the visualization tools.

14.6.1 Using Data Warehouse Information

There are decision support technologies that help utilize the data available in a data warehouse. These technologies help executives to use the warehouse quickly and effectively. They can gather data, analyze it, and take decisions based on the information present in the warehouse. The information gathered in a warehouse can be used in any of the following domains −

- **Tuning Production Strategies** − The product strategies can be well tuned by repositioning the products and managing the product portfolios by comparing the sales quarterly or yearly.
- **Customer Analysis** − Customer analysis is done by analyzing the customer's buying preferences, buying time, budget cycles, etc.
- **Operations Analysis** − Data warehousing also helps in customer relationship management, and making environmental corrections. The information also allows us to analyze business operations.

14.6.2 Integrating Heterogeneous Databases

To integrate heterogeneous databases, we have two approaches −

- Query-driven Approach
- Update-driven Approach

14.7 Query-Driven Approach

This is the traditional approach to integrate heterogeneous databases. This approach was used to build wrappers and integrators on top of multiple heterogeneous databases. These integrators are also known as mediators.

(a) Process of Query-Driven Approach

- When a query is issued to a client side, a metadata dictionary translates the query into an appropriate form for individual heterogeneous sites involved.
- Now these queries are mapped and sent to the local query processor.
- The results from heterogeneous sites are integrated into a global answer set.

(b) Disadvantages

- Query-driven approach needs complex integration and filtering processes.
- This approach is very inefficient.

- It is very expensive for frequent queries.
- This approach is also very expensive for queries that require aggregations.

14.7.1 Update-Driven Approach

This is an alternative to the traditional approach. Today's data warehouse systems follow update-driven approach rather than the traditional approach discussed earlier. In update-driven approach, the information from multiple heterogeneous sources are integrated in advance and are stored in a warehouse. This information is available for direct querying and analysis.

a. Advantages

This approach has the following advantages −

- This approach provide high performance.
- The data is copied, processed, integrated, annotated, summarized and restructured in semantic data store in advance.
- Query processing does not require an interface to process data at local sources.

14.7.2 Functions of Data Warehouse Tools and Utilities

The following are the functions of data warehouse tools and utilities −

- **Data Extraction** − Involves gathering data from multiple heterogeneous sources.
- **Data Cleaning** − Involves finding and correcting the errors in data.
- **Data Transformation** − Involves converting the data from legacy format to warehouse format.
- **Data Loading** − Involves sorting, summarizing, consolidating, checking integrity, and building indices and partitions.
- **Refreshing** − Involves updating from data sources to warehouse.

14.8 Metadata

Metadata is simply defined as data about data. The data that are used to represent other data is known as metadata. For example, the index of a book serves as a metadata for the contents in the book. In other words, we can say that metadata is the summarized data that leads us to the detailed data.

In terms of data warehouse, we can define metadata as following −

- Metadata is a road-map to data warehouse.
- Metadata in data warehouse defines the warehouse objects.
- Metadata acts as a directory. This directory helps the decision support system to locate the contents of a data warehouse.

14.8.1 Metadata Repository

Metadata repository is an integral part of a data warehouse system. It contains the following metadata −

- **Business metadata** − It contains the data ownership information, business definition, and changing policies.
- **Operational metadata** − It includes currency of data and data lineage. Currency of data refers to the data being active, archived, or purged. Lineage of data means history of data migrated and transformation applied on it.
- **Data for mapping from operational environment to data warehouse** − It metadata includes source databases and their contents, data extraction, data partition, cleaning, transformation rules, data refresh and purging rules.
- **The algorithms for summarization** − It includes dimension algorithms, data on granularity, aggregation, summarizing, etc.

14.8.2 Data Cube

A data cube helps us represent data in multiple dimensions. It is defined by dimensions and facts. The dimensions are the entities with respect to which an enterprise preserves the records.

Illustration of Data Cube

Suppose a company wants to keep track of sales records with the help of sales data warehouse with respect to time, item, branch, and location. These dimensions allow to keep track of monthly sales and at which branch the items were sold. There is a table associated with each dimension. This table is known as dimension table. For example, "item" dimension table may have attributes such as item_name, item_type, and item_brand.

The following table represents the 2-D view of Sales Data for a company with respect to time, item, and location dimensions.

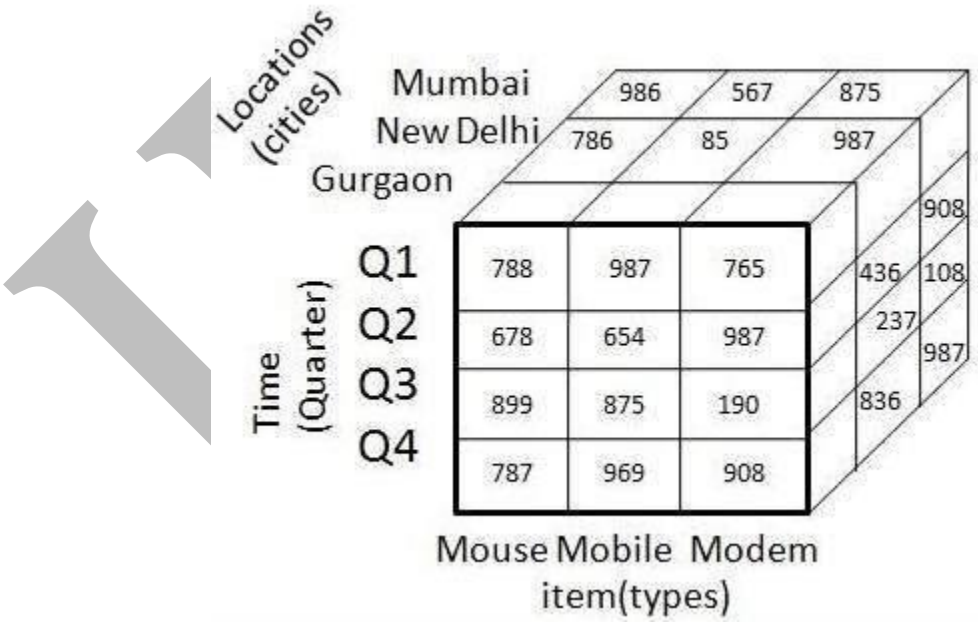
Location="New Delhi"				
Time(quarter)	Item(type)			
	Entertainment	Keyboard	Mobile	Locks
Q1	500	700	10	300
Q2	769	765	30	476
Q3	987	489	18	659
Q4	666	976	40	539

But here in this 2-D table, we have records with respect to time and item only. The sales for New Delhi are shown with respect to time, and item dimensions according to type of items sold. If we want to view

the sales data with one more dimension, say, the location dimension, then the 3-D view would be useful. The 3-D view of the sales data with respect to time, item, and location is shown in the table below −

Time	Location="Gurgaon"			Location="New Delhi"			Location="Mumbai"		
	Item			Item			Item		
	Mouse	Mobile	Modem	Mouse	Mobile	Modem	Mouse	Mobile	Modem
Q1	788	987	765	786	85	987	986	567	875
Q2	678	654	987	659	786	436	980	876	908
Q3	899	875	190	983	909	237	987	100	1089
Q4	787	969	908	537	567	836	837	926	987

The above 3-D table can be represented as 3-D data cube as shown in the following figure −



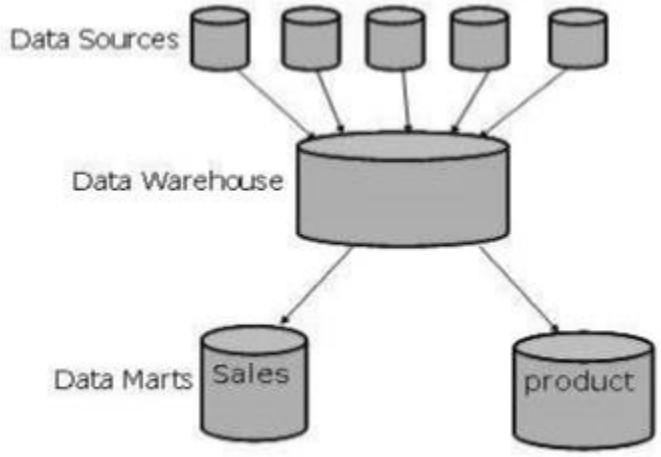
14.9 Data Mart

Data marts contain a subset of organization-wide data that is valuable to specific groups of people in an organization. In other words, a data mart contains only those data that is specific to a particular group. For example, the marketing data mart may contain only data related to items, customers, and sales. Data marts are confined to subjects.

(a) Points to Remember About Data Marts

- Windows-based or Unix/Linux-based servers are used to implement data marts. They are implemented on low-cost servers.
- The implementation cycle of a data mart is measured in short periods of time, i.e., in weeks rather than months or years.
- The life cycle of data marts may be complex in the long run, if their planning and design are not organization-wide.
- Data marts are small in size.
- Data marts are customized by department.
- The source of a data mart is departmentally structured data warehouse.
- Data marts are flexible.

The following figure shows a graphical representation of data marts.



Three-Tier Data Warehouse Architecture

Generally a data warehouses adopts a three-tier architecture. Following are the three tiers of the data warehouse architecture.

- **Bottom Tier** − The bottom tier of the architecture is the data warehouse database server. It is the relational database system. We use the back end tools and utilities to feed data into the bottom tier. These back end tools and utilities perform the Extract, Clean, Load, and refresh functions.
- **Middle Tier** − In the middle tier, we have the OLAP Server that can be implemented in either of the following ways.
    - By Relational OLAP (ROLAP), which is an extended relational database management system. The ROLAP maps the operations on multidimensional data to standard relational operations.
    - By Multidimensional OLAP (MOLAP) model, which directly implements the multidimensional data and operations.

- **Top-Tier** − This tier is the front-end client layer. This layer holds the query tools and reporting tools, analysis tools and data mining tools.

The following diagram depicts the three-tier architecture of data warehouse −

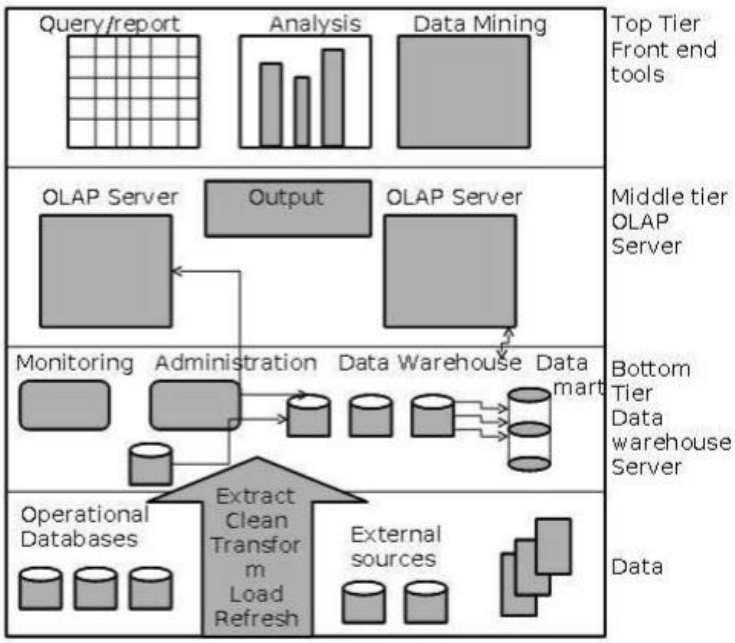


Figure 14.3: 3-tier Datawarehouse architecture

14.10 Types of OLAP Servers

We have four types of OLAP servers −

- Relational OLAP (ROLAP)
- Multidimensional OLAP (MOLAP)
- Hybrid OLAP (HOLAP)
- Specialized SQL Servers

1. Relational OLAP

ROLAP servers are placed between relational back-end server and client front-end tools. To store and manage warehouse data, ROLAP uses relational or extended-relational DBMS.

ROLAP includes the following −

- Implementation of aggregation navigation logic.
- Optimization for each DBMS back end.
- Additional tools and services.

2. Multidimensional OLAP

MOLAP uses array-based multidimensional storage engines for multidimensional views of data. With multidimensional data stores, the storage utilization may be low if the data set is sparse. Therefore, many MOLAP server use two levels of data storage representation to handle dense and sparse data sets.

3. Hybrid OLAP

Hybrid OLAP is a combination of both ROLAP and MOLAP. It offers higher scalability of ROLAP and faster computation of MOLAP. HOLAP servers allows to store the large data volumes of detailed information. The aggregations are stored separately in MOLAP store.

4. Specialized SQL Servers

Specialized SQL servers provide advanced query language and query processing support for SQL queries over star and snowflake schemas in a read-only environment.

5. OLAP Operations

Since OLAP servers are based on multidimensional view of data, we will discuss OLAP operations in multidimensional data.

Here is the list of OLAP operations −

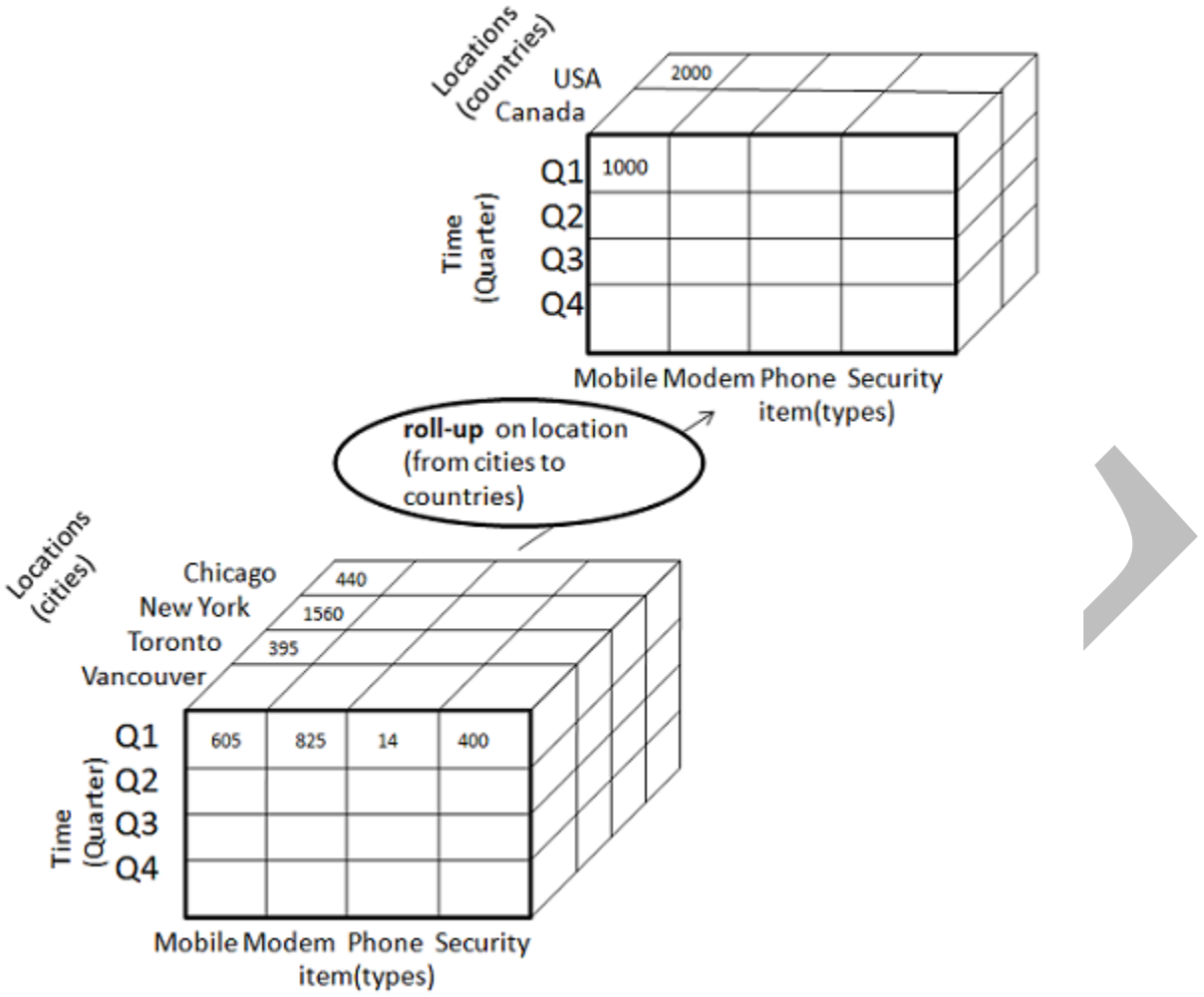
- Roll-up
- Drill-down
- Slice and dice
- Pivot (rotate)

(a) Roll-up

Roll-up performs aggregation on a data cube in any of the following ways −

- By climbing up a concept hierarchy for a dimension
- By dimension reduction

The following diagram illustrates how roll-up works.



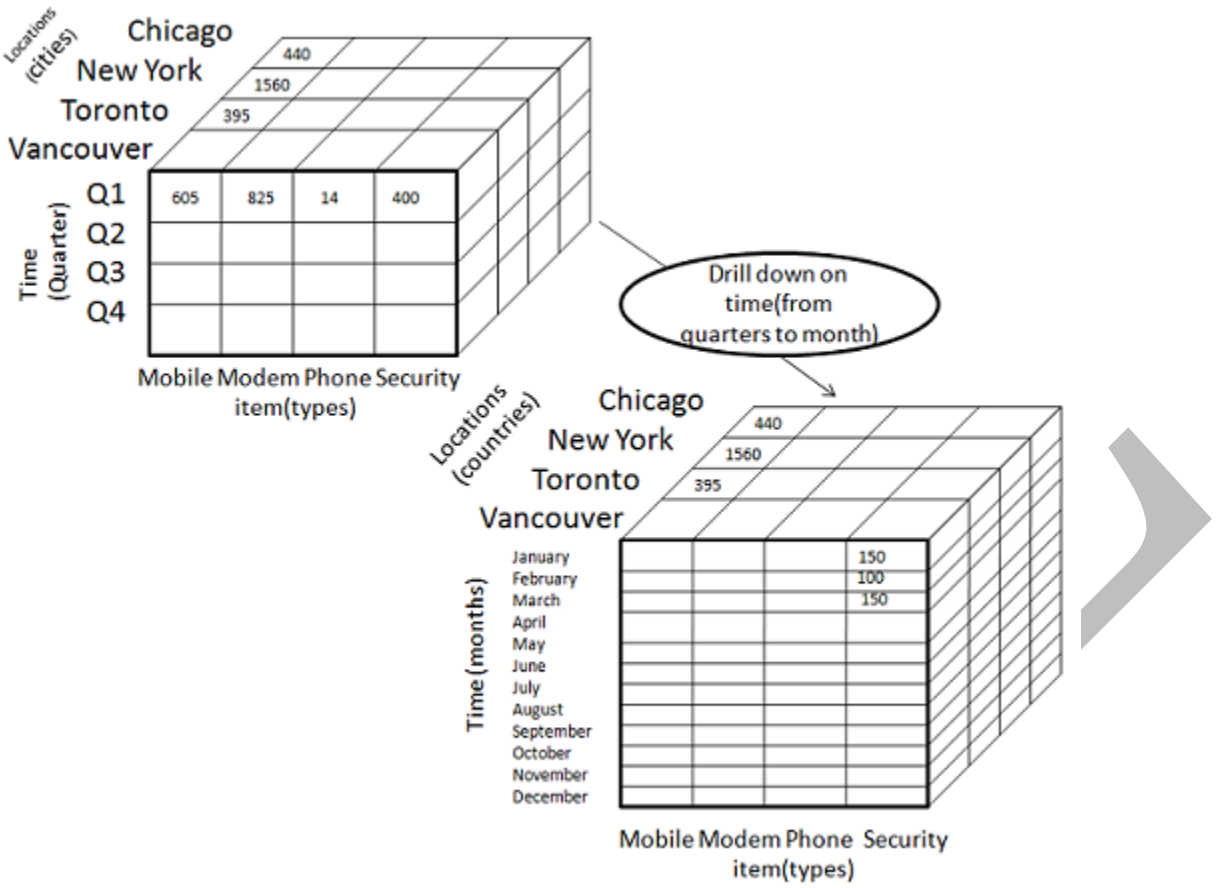
- Roll-up is performed by climbing up a concept hierarchy for the dimension location.
- Initially the concept hierarchy was "street < city < province < country".
- On rolling up, the data is aggregated by ascending the location hierarchy from the level of city to the level of country.
- The data is grouped into cities rather than countries.
- When roll-up is performed, one or more dimensions from the data cube are removed.

(b) Drill-down

Drill-down is the reverse operation of roll-up. It is performed by either of the following ways −

- By stepping down a concept hierarchy for a dimension
- By introducing a new dimension.

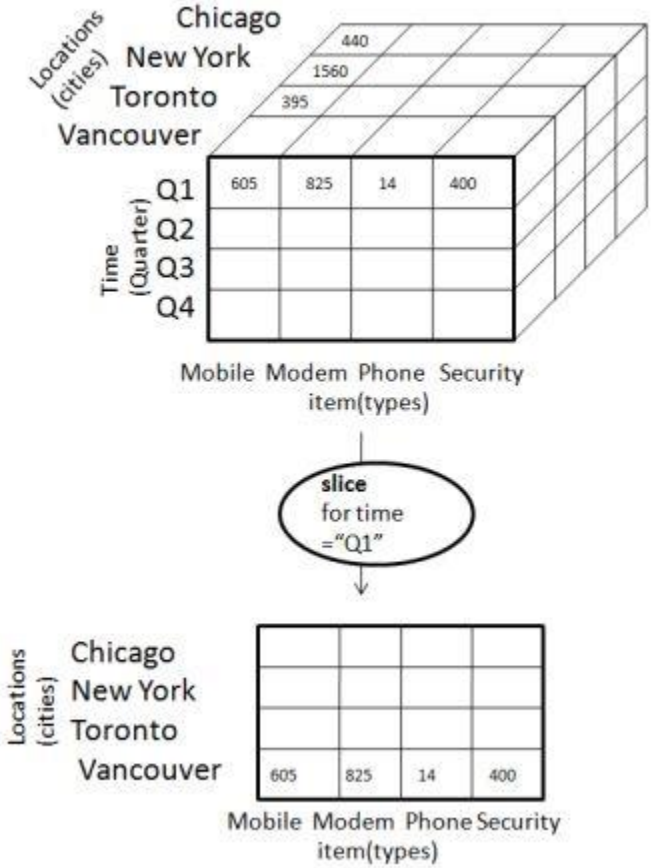
The following diagram illustrates how drill-down works −



- Drill-down is performed by stepping down a concept hierarchy for the dimension time.
- Initially the concept hierarchy was "day < month < quarter < year."
- On drilling down, the time dimension is descended from the level of quarter to the level of month.
- When drill-down is performed, one or more dimensions from the data cube are added.
- It navigates the data from less detailed data to highly detailed data.

(c) Slice

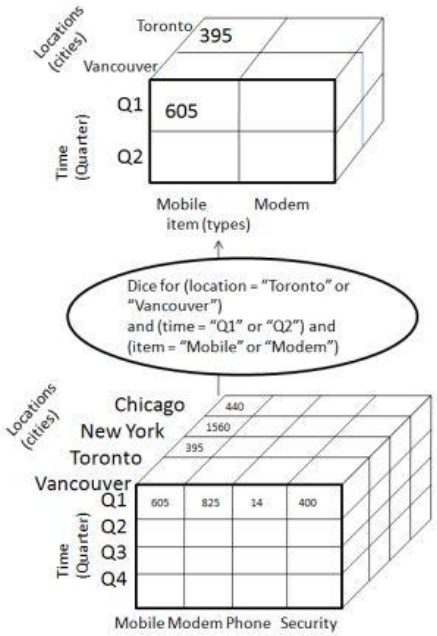
The slice operation selects one particular dimension from a given cube and provides a new sub-cube. Consider the following diagram that shows how slice works.



- Here Slice is performed for the dimension "time" using the criterion time = "Q1".
- It will form a new sub-cube by selecting one or more dimensions.

(d) Dice

Dice selects two or more dimensions from a given cube and provides a new sub-cube. Consider the following diagram that shows the dice operation.



The dice operation on the cube based on the following selection criteria involves three dimensions.

- (location = "Toronto" or "Vancouver")
- (time = "Q1" or "Q2")
- (item =" Mobile" or "Modem")

14.11 OLAP vs OLTP

Data Warehouse (OLAP)	**Operational Database (OLTP)**
Involves historical processing of information.	Involves day-to-day processing.
OLAP systems are used by knowledge workers such as executives, managers and analysts.	OLTP systems are used by clerks, DBAs, or database professionals.
Useful in analyzing the business.	Useful in running the business.
It focuses on Information out.	It focuses on Data in.
Based on Star Schema, Snowflake, Schema and Fact Constellation Schema.	Based on Entity Relationship Model.
Contains historical data.	Contains current data.
Provides summarized and consolidated data.	Provides primitive and highly detailed data.
Provides summarized and multidimensional view of data.	Provides detailed and flat relational view of data.
Number or users is in hundreds.	Number of users is in thousands.
Number of records accessed is in millions.	Number of records accessed is in tens.
Database size is from 100 GB to 1 TB	Database size is from 100 MB to 1 GB.
Highly flexible.	Provides high performance.

14.11.1 Relational OLAP

Relational OLAP servers are placed between relational back-end server and client front-end tools. To store and manage the warehouse data, the relational OLAP uses relational or extended-relational DBMS.

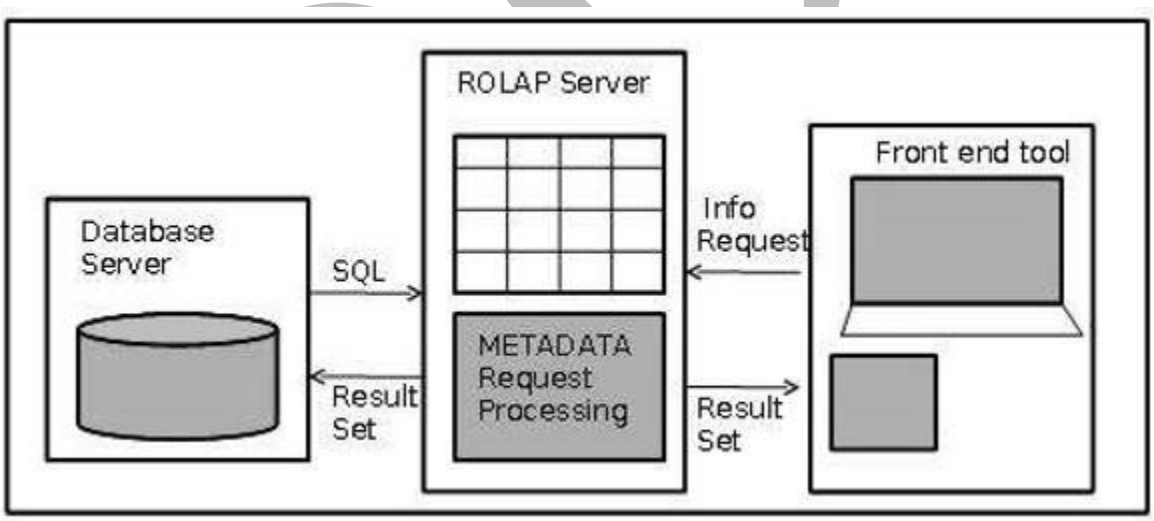
ROLAP includes the following −

- Implementation of aggregation navigation logic
- Optimization for each DBMS back-end
- Additional tools and services
- ROLAP servers are highly scalable.
- ROLAP tools analyze large volumes of data across multiple dimensions.
- ROLAP tools store and analyze highly volatile and changeable data.

14.11.2 Relational OLAP Architecture

ROLAP includes the following components −

- Database server
- ROLAP server
- Front-end tool.



Advantages

- ROLAP servers can be easily used with existing RDBMS.
- Data can be stored efficiently, since no zero facts can be stored.
- ROLAP tools do not use pre-calculated data cubes.
- DSS server of micro-strategy adopts the ROLAP approach.

Disadvantages

- Poor query performance.
- Some limitations of scalability depending on the technology architecture that is utilized.

14.11.3 MOLAP

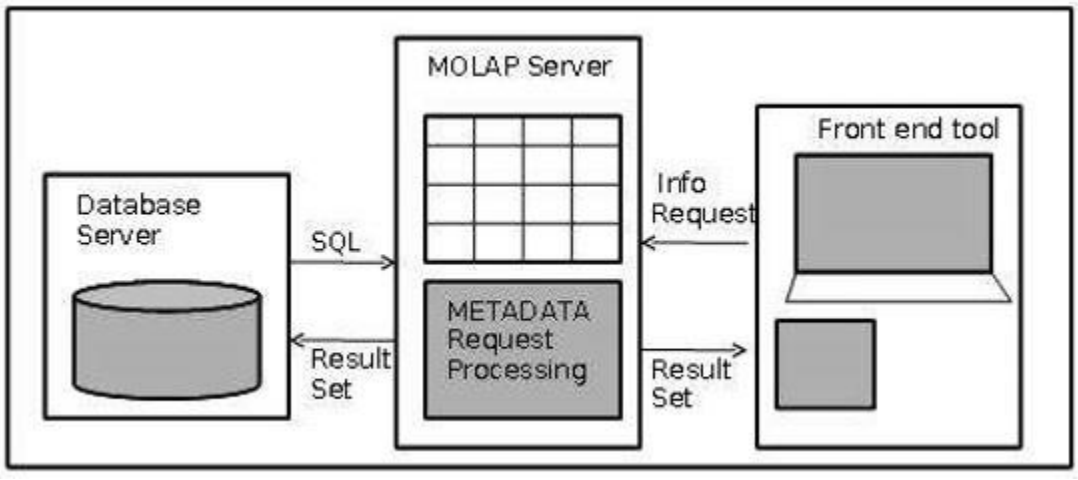
Multidimensional OLAP (MOLAP) uses array-based multidimensional storage engines for multidimensional views of data. With multidimensional data stores, the storage utilization may be low if the dataset is sparse. Therefore, many MOLAP servers use two levels of data storage representation to handle dense and sparse datasets.

- MOLAP tools process information with consistent response time regardless of level of summarizing or calculations selected.
- MOLAP tools need to avoid many of the complexities of creating a relational database to store data for analysis.
- MOLAP tools need fastest possible performance.
- MOLAP server adopts two level of storage representation to handle dense and sparse data sets.
- Denser sub-cubes are identified and stored as array structure.
- Sparse sub-cubes employ compression technology.

14.11.3 MOLAP Architecture

MOLAP includes the following components −

- Database server.
- MOLAP server.
- Front-end tool.



Advantages

- MOLAP allows fastest indexing to the pre-computed summarized data.
- Helps the users connected to a network who need to analyze larger, less-defined data.

- Easier to use, therefore MOLAP is suitable for inexperienced users.

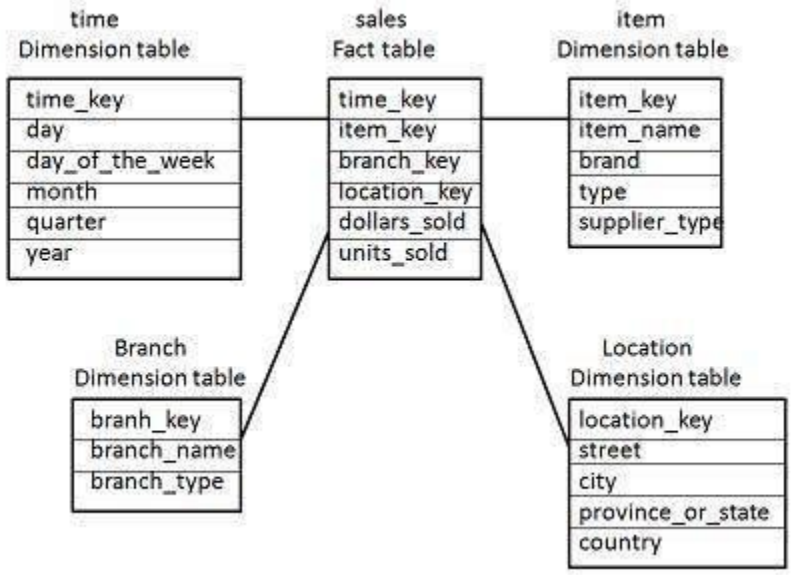
Disadvantages

- MOLAP are not capable of containing detailed data.
- The storage utilization may be low if the data set is sparse.

14.12Star Schema

Schema is a logical description of the entire database. It includes the name and description of records of all record types including all associated data-items and aggregates. Much like a database, a data warehouse also requires to maintain a schema. A database uses relational model, while a data warehouse uses Star, Snowflake, and Fact Constellation schema. In this chapter, we will discuss the schemas used in a data warehouse.

- Each dimension in a star schema is represented with only one-dimension table.
- This dimension table contains the set of attributes.
- The following diagram shows the sales data of a company with respect to the four dimensions, namely time, item, branch, and location.



- There is a fact table at the center. It contains the keys to each of four dimensions.
- The fact table also contains the attributes, namely dollars sold and units sold.

Exercise:

1. What is datawarehouse? Explain its need.
2. Explain the role of datwarehouse in Industry.
3. Explain architecture and components of datawarehouse system.
4. Explain ROLAP and MOLAP with suitable example.
5. What is datamart? Explain.
6. Give comparison between operational system and data warehouse system.

7. Explain the schema used for datawarehouse system.
8. Write a note on Metadata.

Chapter 15
Data Warehousing, Decision Support System and Data Mining

15.1 Introduction

Decision support systems (DSS) are interactive software-based systems intended to help managers in decision-making by accessing large volumes of information generated from various related information systems involved in organizational business processes, such as office automation system, transaction processing system, etc.

DSS uses the summary information, exceptions, patterns, and trends using the analytical models. A decision support system helps in decision-making but does not necessarily give a decision itself. The decision makers compile useful information from raw data, documents, personal knowledge, and/or business models to identify and solve problems and make decisions.

15.2 Datawarehousing

A data warehouse is an information system which stores historical and commutative data from single or multiple sources. It is designed to analyze, report, integrate transaction data from different sources.

Data Warehouse eases the analysis and reporting process of an organization. It is also a single version of truth for the organization for decision making and forecasting process.

Characteristics of Data Warehouse

- A data warehouse is subject oriented as it offers information related to theme instead of companies' ongoing operations.
- The data also needs to be stored in the Datawarehouse in common and unanimously acceptable manner.
- The time horizon for the data warehouse is relatively extensive compared with other operational systems.
- A data warehouse is non-volatile which means the previous data is not erased when new information is entered in it.

15.3 Decision Support System

It is important to understand that when using the DSS, decisions are made by the regulatory agency, not for them. The DSS is a decision support system, not a decision making system. There is no magic here; the regulatory agencies themselves (working with a panel of experts that includes, at least, some on-site wastewater scientists) determine the value of data characteristics. But the DSS process aids in that determination. When the DSS is used as designed to be used, then it can help to assure that there are no unsaid assumptions about the amount of weight assigned to different data quality/quantity attributes. When actual study results are assessed, the DSS helps regulators determine quantitative scores for the data by breaking the scoring of the data down into the eight data attributes. Throughout this entire process the DSS uses a quantitative approach giving quantitative rankings based upon the type of study, as well as quantitative weights for data attributes and numerical data scores for the actual research data.

The DSS assists users in evaluating appropriate rankings for differing types of studies (i.e., datasets). It does not provide the data rankings, but helps regulators and scientists develop their own quantitative rankings for different datasets based upon their understanding of the research process and perceptions (biases) regarding which attributes of the data are most valuable for regulatory decision-making. An expert panel approach is used wherein a series of questionnaires in the form of Excel spreadsheets are given to the interested parties (onsite technology regulators and onsite wastewater scientists in this case). The DSS approach provides a self-assessment weighting tool to facilitate determining the value of different types of data quality and quantity attributes.

15.3.1 Programmed and Non-programmed Decisions

There are two types of decisions - programmed and non-programmed decisions.

Programmed decisions are basically automated processes, general routine work, where −

- These decisions have been taken several times.
- These decisions follow some guidelines or rules.

For example, selecting a reorder level for inventories, is a programmed decision.

Non-programmed decisions occur in unusual and non-addressed situations, so −

- It would be a new decision.
- There will not be any rules to follow.
- These decisions are made based on the available information.
- These decisions are based on the manger's discretion, instinct, perception and judgment.

For example, investing in a new technology is a non-programmed decision.

Decision support systems generally involve non-programmed decisions. Therefore, there will be no exact report, content, or format for these systems. Reports are generated on the fly.

15.3.2 Attributes of a DSS

- Adaptability and flexibility
- High level of Interactivity
- Ease of use
- Efficiency and effectiveness
- Complete control by decision-makers
- Ease of development
- Extendibility
- Support for modeling and analysis
- Support for data access
- Standalone, integrated, and Web-based

15.3.3 Characteristics of a DSS

- Support for decision-makers in semi-structured and unstructured problems.
- Support for managers at various managerial levels, ranging from top executive to line managers.
- Support for individuals and groups. Less structured problems often requires the involvement of several individuals from different departments and organization level.
- Support for interdependent or sequential decisions.
- Support for intelligence, design, choice, and implementation.
- Support for variety of decision processes and styles.
- DSSs are adaptive over time.

15.3.4 Benefits of DSS

- Improves efficiency and speed of decision-making activities.
- Increases the control, competitiveness and capability of futuristic decision-making of the organization.
- Facilitates interpersonal communication.
- Encourages learning or training.
- Since it is mostly used in non-programmed decisions, it reveals new approaches and sets up new evidences for an unusual decision.
- Helps automate managerial processes.

DSS have been classified in different ways as the concept matured with time. As. and when the full potential and possibilities for the field emerged, different classification systems also emerged. Some of the well known classification models are given below:

- According to Donovan and Madnick (1977) DSS can be classified as,

1. Institutional-when the DSS supports ongoing and recurring decisions
2. Ad hoc-when the DSS supports a one off-kind of decision.

- Hackathorn and Keen (1981) classified DSS as,

    1. Personal DSS

    2. Group DSS

    3. Organizational DSS

- Alter (1980) opined that decision support systems could be classified into seven types based on their generic nature of operations. He described the seven types as,

1. File drawer systems. This type of DSS primarily provides access to data stores/data related items.
2. Data analysis systems. This type of DSS supports the manipulation of data through the use of specific or generic computerized settings or tools.
3. Analysis information systems. This type of DSS provides access to sets of decision oriented databases and simple small models.

4. Accounting and financial models. This type of DSS can perform 'what if analysis' and calculate the outcomes of different decision paths.
5. Representational models. This type of DSS can also perform 'what if analysis' and calculate the outcomes of different decision paths, based on simulated models.
6. Optimization models. This kind of DSS provides solutions through the use of optimization models which have mathematical solutions.
7. Suggestion models. This kind of DSS works when the decision to be taken is based on well-structured tasks.

Modern classification of DSS are,

1. Model Driven DSS is a DSS that uses a model (quantitative) based on heuristics, optimization, simulation etc. for deriving solutions to problems. It has access to the models and has flexibility of changing the parameters of the model. Real data or transactional data from databases of TPS is then passed through the model to arrive at the solution. The system is capable of producing different scenarios.
2. Data Driven DSS is a DSS that gives access to time-series internal data. Data ware houses that have tools that provide facility to manipulate such data are examples of advances systems. Executive Information Systems are examples of data-driven DSS.
3. Communications-driven DSS is a DSS that uses network and communications technologies to support decision-relevant collaboration and communication. In such systems, communication technologies are the most important component.
4. Document-driven DSS is a DSS that uses computer storage and processing to provide document retrieval and analysis.
5. Knowledge-driven DSS is a DSS that collects and stores 'expertise' so that it can be used for decision-making when required.

Components of DSS

Even though DSS can be of several types, fundamentally each DSS will have the following components:

- Interactive User-System Dialog Management Subsystem-DSS requires continuous user interaction. Sometimes the system should prompt the user to give an input at other time the user should be able to control the processing. A typical user system dialog management subsystem will have the following elements:

1. User Interface - the user interface of a DSS has to be dynamic and GUI based. It has to be an easy to use user interface as most of the people who will be using it are not technical experts but management experts (top management) and hence the interface should be minimalist in design. Also the system should be able to interact with the user in a interactive mode and hence the user interface has to be dynamic.
2. Request Constructor - since DSS works on an interactive dynamic mode, it needs a request constructor (incorporating aspects of Language Query Interface) which can convert the user's instructors into model understandable form, the model's data request to the database and the model's instructions/requests to the user.

Data Management Subsystem - data is the most important component of a DSS. Without the data a DSS cannot function. The data management subsystem manages the data for DSS. Data is accessed in a DSS in many ways like ad hoc basis, structured query basis and heuristic search basis and hence a strong data management subsystems is required to service the varied data requests from a DSS. The subsystem has the following elements:

1. Database Management System - it is the data store for the DSS. It manages the data and performs all the functions that a typical DBMS package does. In fact, in most DSS a commercial DBMS or RDBMS package is used to perform this task.
2. The Query Control - this is a tailored element to handle the query requirements of DSS. It may connect the database, directly to the user interface or to the model base or both.
3. Meta Data - this contains data about the data that is stored in the database. This helps the DSS in understanding the data in the database properly and helps in creating ad hoc queries.

- Model Management Subsystem - this is the unique feature of a DSS. This makes the system special. However, this also makes the system very specific. There are very few examples of a generalized DSS as generalized models are not available. Those that exist work on half baked solutions. The model management subsystem may use different classes of models like,

1. Optimization Models
2. Simulation Models
3. Heuristic Models
4. Deterministic Models
5. Predictive Models

Each class of model is useful to solve a specific class of problems like a routing problem or a scheduling problem or a combinatorial search problem etc. Model and Model Management has several connotations in DSS literature and there have been wide ranging definitions of these terms. The common strain that evolves from these plethora of definitions is that a model is conceived to consist of a solver, a model for solving a problem and data (Ramirez, 1993) where model represents relationships between variables, data represents the values of the variables under consideration and the solver is the tool that enables the computation of the variable values and their relationships. It has been also conceptualized in some literature as a procedure which works on the data to give an output after analysis.

The model management subsystem has the following elements:

1. The Model Base Management System-A model base or rather a model base management system is software is conceptually like what the DBMS is to data which has the capabilities to manage a model for it to be useful to the decision maker. It is the core of a DSS. It supports generation of models and works with data on one hand and the user supplied instructions on the other.
2. The Model Command Processor-is the entity that processes the commands coming from the dialog management subsystem.
3. The Model Executor or Solver- is the heart of the system. It is the process through which the model is solved using some algorithm. It works with the model as generated by the model base with instructions from the user, the request constructor (dialog management subsystem in

general) to get the parameters of the model from the user and data from the data management subsystem. It then solves the problem and displays the results and some variations of the best fit solution through the dialog management subsystem. The alternative solutions as provided help the user in decision-making.

15.4 Types of DSS

Following are some typical DSSs −

- **Status Inquiry System** − It helps in taking operational, management level, or middle level management decisions, for example daily schedules of jobs to machines or machines to operators.
- **Data Analysis System** − It needs comparative analysis and makes use of formula or an algorithm, for example cash flow analysis, inventory analysis etc.
- **Information Analysis System** − In this system data is analyzed and the information report is generated. For example, sales analysis, accounts receivable systems, market analysis etc.
- **Accounting System** − It keeps track of accounting and finance related information, for example, final account, accounts receivables, accounts payables, etc. that keep track of the major aspects of the business.
- **Model Based System** − Simulation models or optimization models used for decision-making are used infrequently and creates general guidelines for operation or management.

All the systems we are discussing here come under knowledge management category. A knowledge management system is not radically different from all these information systems, but it just extends the already existing systems by assimilating more information.

As we have seen, data is raw facts, information is processed and/or interpreted data, and knowledge is personalized information.

15.5 What is Knowledge?

- Personalized information
- State of knowing and understanding
- An object to be stored and manipulated
- A process of applying expertise
- A condition of access to information
- Potential to influence action

15.6 Sources of Knowledge of an Organization

- Intranet
- Data warehouses and knowledge repositories
- Decision support tools
- Groupware for supporting collaboration
- Networks of knowledge workers
- Internal expertise

15.6.1 Definition of KMS

A knowledge management system comprises a range of practices used in an organization to identify, create, represent, distribute, and enable adoption to insight and experience. Such insights and experience comprise knowledge, either embodied in individual or embedded in organizational processes and practices.

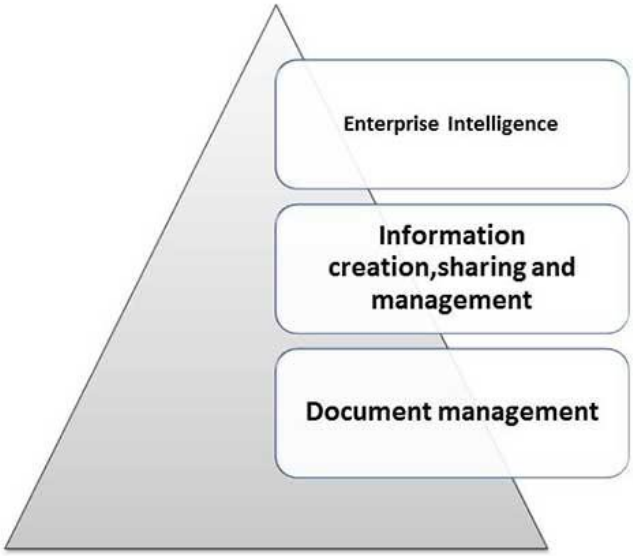
Purpose of KMS

- Improved performance
- Competitive advantage
- Innovation
- Sharing of knowledge
- Integration
- Continuous improvement by −
    - Driving strategy
    - Starting new lines of business
    - Solving problems faster
    - Developing professional skills
    - Recruit and retain talent

15.6.2 Activities in Knowledge Management

- Start with the business problem and the business value to be delivered first.
- Identify what kind of strategy to pursue to deliver this value and address the KM problem.
- Think about the system required from a people and process point of view.
- Finally, think about what kind of technical infrastructure are required to support the people and processes.
- Implement system and processes with appropriate change management and iterative staged release.

15.7 Level of Knowledge Management



Executive support systems are intended to be used by the senior managers directly to provide support to non-programmed decisions in strategic management.

These information are often external, unstructured and even uncertain. Exact scope and context of such information is often not known beforehand.

This information is intelligence based −

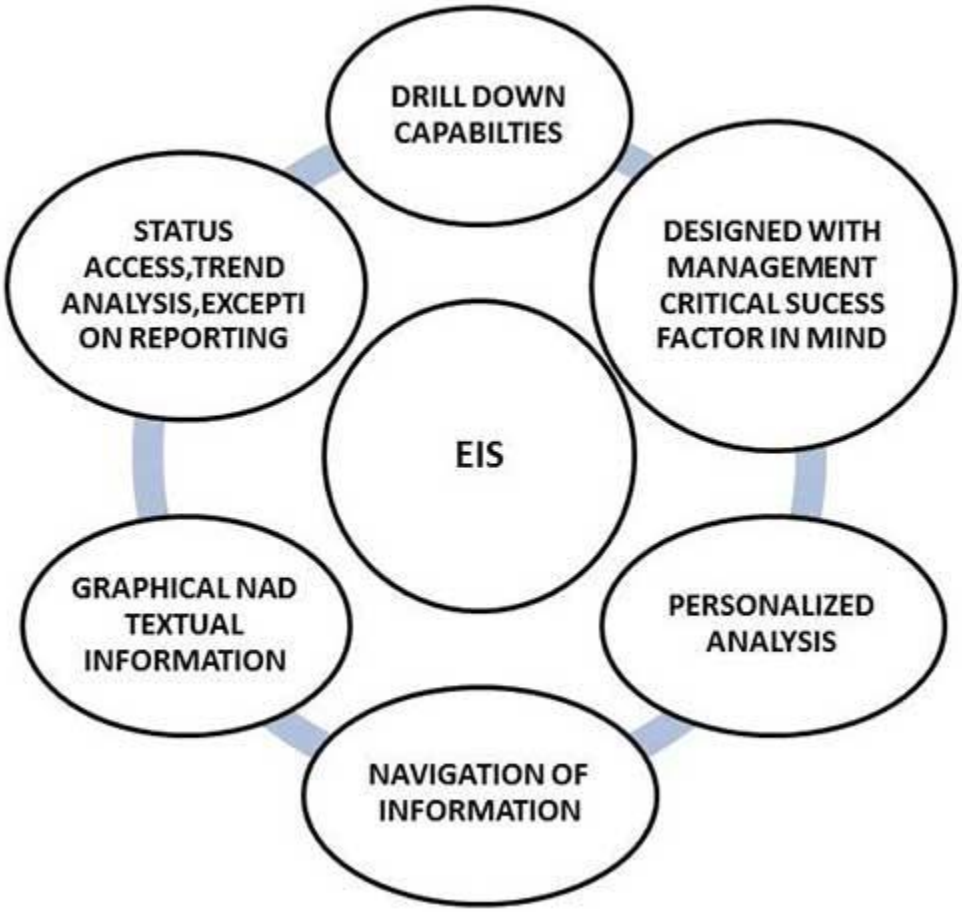
- Market intelligence
- Investment intelligence
- Technology intelligence

Examples of Intelligent Information

Following are some examples of intelligent information, which is often the source of an ESS −

- External databases
- Technology reports like patent records etc.
- Technical reports from consultants
- Market reports
- Confidential information about competitors
- Speculative information like market conditions
- Government policies
- Financial reports and information

15.7 Features of Executive Information System



15.7.2 Advantages of ESS

- Easy for upper level executive to use
- Ability to analyze trends
- Augmentation of managers' leadership capabilities
- Enhance personal thinking and decision-making
- Contribution to strategic control flexibility
- Enhance organizational competitiveness in the market place
- Instruments of change
- Increased executive time horizons.
- Better reporting system
- Improved mental model of business executive
- Help improve consensus building and communication
- Improve office automation
- Reduce time for finding information
- Early identification of company performance
- Detail examination of critical success factor
- Better understanding
- Time management

- Increased communication capacity and quality

15.7.3 Disadvantage of ESS

- Functions are limited
- Hard to quantify benefits
- Executive may encounter information overload
- System may become slow
- Difficult to keep current data
- May lead to less reliable and insecure data
- Excessive cost for small company

The term 'Business Intelligence' has evolved from the decision support systems and gained strength with the technology and applications like data warehouses, Executive Information Systems and Online Analytical Processing (OLAP).

Business Intelligence System is basically a system used for finding patterns from existing data from operations.

15.7.4 Characteristics of BIS

- It is created by procuring data and information for use in decision-making.
- It is a combination of skills, processes, technologies, applications and practices.
- It contains background data along with the reporting tools.
- It is a combination of a set of concepts and methods strengthened by fact-based support systems.
- It is an extension of Executive Support System or Executive Information System.
- It collects, integrates, stores, analyzes, and provides access to business information
- It is an environment in which business users get reliable, secure, consistent, comprehensible, easily manipulated and timely information.
- It provides business insights that lead to better, faster, more relevant decisions.

15.7.5 Benefits of BIS

- Improved Management Processes.
- Planning, controlling, measuring and/or applying changes that results in increased revenues and reduced costs.
- Improved business operations.
- Fraud detection, order processing, purchasing that results in increased revenues and reduced costs.
- Intelligent prediction of future.

15.7.6 Approaches of BIS

For most companies, it is not possible to implement a proactive business intelligence system at one go. The following techniques and methodologies could be taken as approaches to BIS −

- Improving reporting and analytical capabilities
- Using scorecards and dashboards
- Enterprise Reporting
- On-line Analytical Processing (OLAP) Analysis
- Advanced and Predictive Analysis
- Alerts and Proactive Notification
- Automated generation of reports with user subscriptions and "alerts" to problems and/or opportunities.

15.7.7 Capabilities of BIS

- Data Storage and Management −
  - Data ware house
  - Ad hoc analysis
  - Data quality
  - Data mining
- Information Delivery
  - Dashboard
  - Collaboration /search
  - Managed reporting
  - Visualization
  - Scorecard
- Query, Reporting and Analysis
  - Ad hoc Analysis
  - Production reporting
  - OLAP analysis

Data Mining

- Data mining is the process of **extracting the useful information**, which is stored in the large database.
- It is a powerful tool, which is useful for organizations to retrieve the useful information from available data warehouses.
- Data mining can be applied to relational databases, object-oriented databases, data warehouses, structured-unstructured databases, etc.
- Data mining is used in numerous areas like banking, insurance companies, pharmaceutical companies etc.

15.8 Patterns in Data Mining

1. Association

The items or objects in relational databases, transactional databases or any other information repositories are considered, while finding **associations or correlations.**

2. Classification

- The goal of classification is to construct a model with the help of historical data that can accurately predict the value.
- It maps the data into the predefined groups or classes and searches for the new patterns.

For example:
To predict weather on a particular day will be categorized into - sunny, rainy, or cloudy.

3. Regression

- Regression creates predictive models. Regression analysis is used to make predictions based on existing data by applying formulas.
- Regression is very useful for finding (or predicting) the information on the basis of previously known information.

4. Cluster analysis

- It is a process of portioning a set of data into a set of meaningful subclass, called as cluster.
- It is used to place the data elements into the related groups without advanced knowledge of the group definitions.

5. Forecasting
Forecasting is concerned with the discovery of knowledge or information patterns in data that can lead to reasonable predictions about the future.

15.9 Technologies used in data mining

Several techniques used in the development of data mining methods. **Some of them are mentioned below:**

Statistics:

- It uses the mathematical analysis to express representations, model and summarize empirical data or real world observations.
- Statistical analysis involves the collection of methods, applicable to large amount of data to conclude and report the trend.

Machine learning

- **Arthur Samuel** defined machine learning as a field of study that gives computers the ability to learn without being programmed.
- When the new data is entered in the computer, algorithms help the data to grow or change due to machine learning.
- In machine learning, an algorithm is constructed to predict the data from the available database **(Predictive analysis).**
- It is related to computational statistics.

The four types of machine learning are:

1. Supervised learning

- It is based on the classification.
- It is also called as **inductive learning**. In this method, the desired outputs are included in the training dataset.

2. Unsupervised learning

Unsupervised learning is based on clustering. Clusters are formed on the basis of similarity measures and desired outputs are not included in the training dataset.

3. Semi-supervised learning

Semi-supervised learning includes some desired outputs to the training dataset to generate the appropriate functions. This method generally avoids the large number of labeled examples (i.e. desired outputs) .

4. Active learning

- Active learning is a powerful approach in analyzing the data efficiently.
- The algorithm is designed in such a way that, the desired output should be decided by the algorithm itself (the user plays important role in this type).

Information retrieval

Information deals with uncertain representations of the semantics of objects (text, images).
For example: Finding relevant information from a large document.

Database systems and data warehouse

- Databases are used for the purpose of recording the data as well as data warehousing.
- Online Transactional Processing (OLTP) uses databases for day to day transaction purpose.
- To remove the redundant data and save the storage space, data is normalized and stored in the form of tables.
- **E**ntity-**R**elational modeling techniques are used for relational database management system design.
- Data warehouses are used to store historical data which helps to take strategical decision for business.
- It is used for online analytical processing (OALP), which helps to analyze the data.

Decision support system

- Decision support system is a category of information system. It is very useful in decision making for organizations.
- It is an interactive software based system which helps decision makers to extract useful information from the data, documents to make the decision.

Exercise:

1. What is datawarehouse? Discuss its importance.
2. What is decision support system?
3. State advantages and dis-advantages of decision support system.
4. Explain need of decision support system.
5. Give comparison between decision support system and data warehouse system.
6. Explain the relevance of decision support system with data mining.

Chapter 16
Data Mining and Its Applications

16.1 Introduction

Data Mining is defined as the procedure of extracting information from huge sets of data. There is a huge amount of data available in the Information Industry. This data is of no use until it is converted into useful information. It is necessary to analyze this huge amount of data and extract useful information from it.Extraction of information is not the only process we need to perform; data mining also involves other processes such as Data Cleaning, Data Integration, Data Transformation, Data Mining, Pattern Evaluation and Data Presentation. Data Mining is an analytic procedure intended to process data – usually an enormous data– to look for predictable patterns or potential methodical relationships amongst the data. Then these pattern discoveries can be used to extract new knowledge and insights by applying the identified patterns to new subsets of data.

16.2 Datamining concept and technique

Data Mining is defined as extracting information from huge sets of data. In other words, we can say that data mining is the procedure of mining knowledge from data. The information or knowledge extracted so can be used for any of the following applications −

- Market Analysis
- Fraud Detection
- Customer Retention
- Production Control
- Science Exploration

While large-scale information technology has been evolving separate transaction and analytical systems, data mining provides the link between the two. Data mining software analyzes relationships and patterns in stored transaction data based on open-ended user queries. Several types of analytical software are available: statistical, machine learning, and neural networks. Generally, any of four types of relationships are sought:

- **Classes**: Stored data is used to locate data in predetermined groups. For example, a restaurant chain could mine customer purchase data to determine when customers visit and what they typically order. This information could be used to increase traffic by having daily specials.

- **Clusters**: Data items are grouped according to logical relationships or consumer preferences. For example, data can be mined to identify market segments or consumer affinities.

- **Associations**: Data can be mined to identify associations. The beer-diaper example is an example of associative mining.

- **Sequential patterns**: Data is mined to anticipate behavior patterns and trends. For example, an outdoor equipment retailer could predict the likelihood of a backpack being purchased based on a consumer's purchase of sleeping bags and hiking shoes.

Data mining consists of five major elements:

- Extract, transform, and load transaction data onto the data warehouse system.
- Store and manage the data in a multidimensional database system.
- Provide data access to business analysts and information technology professionals.
- Analyze the data by application software.
- Present the data in a useful format, such as a graph or table.

Different levels of analysis are available:

- **Artificial neural networks**: Non-linear predictive models that learn through training and resemble biological neural networks in structure.
- **Genetic algorithms**: Optimization techniques that use processes such as genetic combination, mutation, and natural selection in a design based on the concepts of natural evolution.
- **Decision trees**: Tree-shaped structures that represent sets of decisions. These decisions generate rules for the classification of a dataset. Specific decision tree methods include Classification and Regression Trees (CART) and Chi Square Automatic Interaction Detection (CHAID) . CART and CHAID are decision tree techniques used for classification of a dataset. They provide a set of rules that you can apply to a new (unclassified) dataset to predict which records will have a given outcome. CART segments a dataset by creating 2-way splits while CHAID segments using chi square tests to create multi-way splits. CART typically requires less data preparation than CHAID.
- **Nearest neighbor method**: A technique that classifies each record in a dataset based on a combination of the classes of the *k* record(s) most similar to it in a historical dataset (where *k* 1). Sometimes called the *k*-nearest neighbor technique.
- **Rule induction**: The extraction of useful if-then rules from data based on statistical significance.
- **Data visualization**: The visual interpretation of complex relationships in multidimensional data. Graphics tools are used to illustrate data relationships.

Data mining involves six common classes of tasks:

- Anomaly detection (Outlier/change/deviation detection) – The identification of unusual data records, that might be interesting or data errors and require further investigation.
- Association rule learning (Dependency modeling) – Searches for relationships between variables. For example a supermarket might gather data on customer purchasing habits. Using association rule learning, the supermarket can determine which products are frequently bought

together and use this information for marketing purposes. This is sometimes referred to as market basket analysis.

- Clustering – is the task of discovering groups and structures in the data that are in some way or another "similar", without using known structures in the data.
- Classification – is the task of generalizing known structure to apply to new data. For example, an e-mail program might attempt to classify an e-mail as "legitimate" or as "spam".
- Regression – Attempts to find a function which models the data with the least error.
- Summarization – providing a more compact representation of the data set, including visualization and report generation.

16.2.1 Working Mechanism

Data mining methods can be performed from any source in which data is saved like spreadsheets, flat files, database tables, or any other storage format. The crucial criteria for the information are not the format of the storage, but rather its relevance to the issue to understand it well.

Appropriate data cleansing and arrangement are essential for mining the data. Data mining may use a number of techniques including machine learning, database management, statistical analysis etc.

Some of the best data mining techniques discussed as follows:

1. Prediction

Prediction is amongst the most common techniques for mining the data since it's utilized to forecast the future scenarios based on the current and new data. In predictive data mining – existing & historical data is analysed to identify patterns. Once the patterns are analysed – new data is then fed to these patterns to forecast the future scenarios. Predictive data mining is the most widely recognized class of mining process because it has the most immediate business applications.

2. Classification

Classification is another important technique for data mining. In classification – different techniques are used to classify the data into predefined segments or classes. Classification uses complicated techniques for mining the data to extract different attributes together into clear discernible classes. Classification then would employ techniques and algorithms to the new data to decide to which class this data should belong to. One of the most common example is how Gmail classes the new emails into spam or not spam based on different attributes of the email.

3.Regression

Regression analysis is a procedure of recognizing and breaking down the relationship amongst the different variables. In simple words – regression is a technique to predict various possible outcomes in

different scenarios. Outcome that is needed to be predicted is a dependent variable dependent on scenarios or independent variables.

4. Clustering

Clustering analysis is the technique used to distinguishing data sets that are like one another, to comprehend the similarities and distinctions in the existing and new data. Clusters share certain similar features that can be utilized to develop targeting algorithms. For instance, clusters of buyers with comparable purchasing behavior can be targeted with the same products/services to boost the conversation rate and sales. An outcome from a clustering analysis technique can be the formation of personas created to represent the distinctive customer types in a targeted statistic, demeanor or potential behavior set that may utilize a product, brand, or website correspondingly. Difference between clustering and classification is that while in classification there are predefined classes, in clustering the clusters or classes evolve from the data after it is mined.

5. Association Rule

Association rule detection is a critical interpretive strategy in the **data mining and analysis** process. This method finds the relationship between at least two products. It sees the concealed patterns in the data sets which is utilized to recognize the variables and the continuous event of multiple variables that show up with the most significant frequencies.

It's a fundamental technique; however, you'd be astonished how much knowledge and understanding it can give — the sort of data numerous organizations uses once a day to enhance effectiveness and generate more revenue.

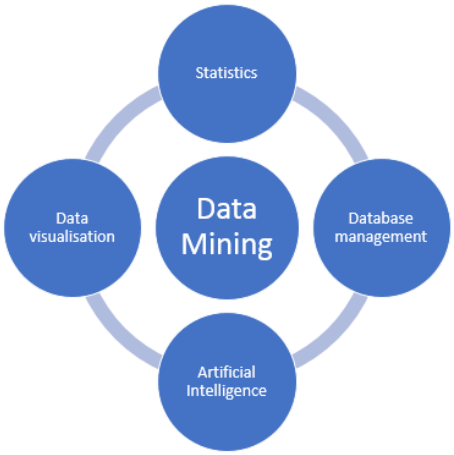


Figure 16.1 Data Mining

Data mining is the amalgamation of statistics, database management, artificial intelligence, machine learning technologies and data visualization. Data mining profession is all about solving this equation: how to prepare and form judgments from vast amounts of data. All the above data mining methods can help in analyzing the unique information from alternate points of view.

16.3 Data Mining Applications

Data mining is highly useful in the following domains −

- Market Analysis and Management
- Corporate Analysis & Risk Management
- Fraud Detection

Apart from these, data mining can also be used in the areas of production control, customer retention, science exploration, sports, astrology, and Internet Web Surf-Aid

16.3.1 Market Analysis and Management

Listed below are the various fields of market where data mining is used −

- **Customer Profiling** − Data mining helps determine what kind of people buy what kind of products.
- **Identifying Customer Requirements** − Data mining helps in identifying the best products for different customers. It uses prediction to find the factors that may attract new customers.
- **Cross Market Analysis** − Data mining performs Association/correlations between product sales.
- **Target Marketing** − Data mining helps to find clusters of model customers who share the same characteristics such as interests, spending habits, income, etc.
- **Determining Customer purchasing pattern** − Data mining helps in determining customer purchasing pattern.
- **Providing Summary Information** − Data mining provides us various multidimensional summary reports.

16.4 Data Mining Terminologies

1. Knowledge Base

This is the domain knowledge. This knowledge is used to guide the search or evaluate the interestingness of the resulting patterns.

2. Knowledge Discovery

Some people treat data mining same as knowledge discovery, while others view data mining as an essential step in the process of knowledge discovery. Here is the list of steps involved in the knowledge discovery process −

- Data Cleaning
- Data Integration
- Data Selection
- Data Transformation
- Data Mining
- Pattern Evaluation
- Knowledge Presentation

3. User interface

User interface is the module of data mining system that helps the communication between users and the data mining system. User Interface allows the following functionalities −

- Interact with the system by specifying a data mining query task.
- Providing information to help focus the search.
- Mining based on the intermediate data mining results.
- Browse database and data warehouse schemas or data structures.
- Evaluate mined patterns.
- Visualize the patterns in different forms.

4. Data Integration

Data Integration is a data preprocessing technique that merges the data from multiple heterogeneous data sources into a coherent data store. Data integration may involve inconsistent data and therefore needs data cleaning.

5. Data Cleaning

Data cleaning is a technique that is applied to remove the noisy data and correct the inconsistencies in data. Data cleaning involves transformations to correct the wrong data. Data cleaning is performed as a data preprocessing step while preparing the data for a data warehouse.

6. Data Selection

Data Selection is the process where data relevant to the analysis task are retrieved from the database. Sometimes data transformation and consolidation are performed before the data selection process.

7. Clusters

Cluster refers to a group of similar kind of objects. Cluster analysis refers to forming group of objects that are very similar to each other but are highly different from the objects in other clusters.

8. Data Transformation

In this step, data is transformed or consolidated into forms appropriate for mining, by performing summary or aggregation operations.

16.5 KDD and Data mining

- The process of discovering knowledge in data and application of data mining techniques are referred to as **Knowledge Discovery in Databases (KDD).**
- KDD consists of various application domains such as artificial intelligence, pattern recognition, machine learning and data visualization.
- The main goal of KDD is to extract knowledge from large databases with the help of data mining methods.

The different steps of KDD are as given below:

1. Data cleaning: In this step, noise and irrelevant data are removed from the database.

2. Data integration: In this step, the heterogeneous data sources are merged into a single data source.

3. Data selection: In this step, the data which is relevant to the analysis process gets retrieved from the database.

4. Data transformation: In this step, the selected data is transformed in such forms which are suitable for data mining.

5. Data mining: In this step, the various techniques are applied to extract the data patterns.

6. Pattern evaluation: In this step, the different data patterns are evaluated.

7. Knowledge representation: This is the final step of KDD, which represents the knowledge.

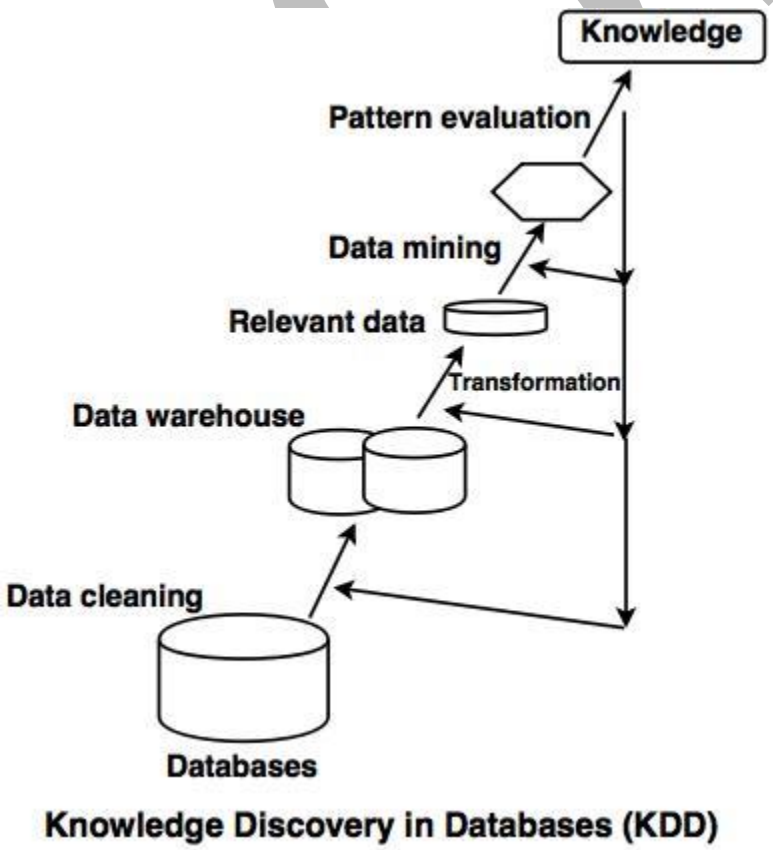


Figure 16.2: KDD Process

Some people don't differentiate data mining from knowledge discovery while others view data mining as an essential step in the process of knowledge discovery.

KDD	Data Mining
It is a field of computer science that helps to extract useful and previously undiscovered knowledge from the large database by using various tools and theories.	**Data mining is one of the important steps in the KDD process.** It includes suitable algorithm based on the objective of the KDD process to identify the patterns from the database.

16.6 Data Mining Techniques

Classification models predict categorical class labels; and prediction models predict continuous valued functions. For example, we can build a classification model to categorize bank loan applications as either safe or risky, or a prediction model to predict the expenditures in dollars of potential customers on computer equipment given their income and occupation.

1 Classification

Following are the examples of cases where the data analysis task is Classification −

- A bank loan officer wants to analyze the data in order to know which customer (loan applicant) are risky or which are safe.
- A marketing manager at a company needs to analyze a customer with a given profile, who will buy a new computer.

In both of the above examples, a model or classifier is constructed to predict the categorical labels. These labels are risky or safe for loan application data and yes or no for marketing data.

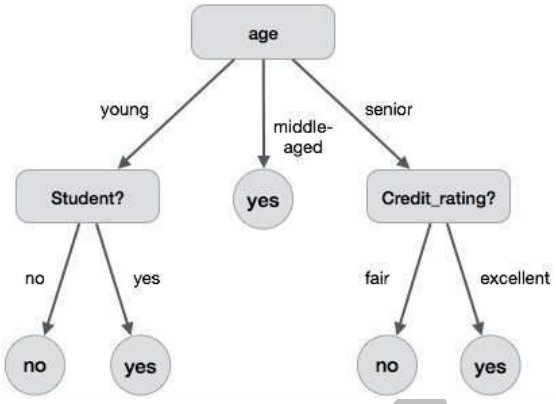
2 Prediction

Following are the examples of cases where the data analysis task is Prediction −

Suppose the marketing manager needs to predict how much a given customer will spend during a sale at his company. In this example we are bothered to predict a numeric value. Therefore the data analysis task is an example of numeric prediction. In this case, a model or a predictor will be constructed that predicts a continuous-valued-function or ordered value.

3 Decision Tree

A decision tree is a structure that includes a root node, branches, and leaf nodes. Each internal node denotes a test on an attribute, each branch denotes the outcome of a test, and each leaf node holds a class label. The topmost node in the tree is the root node.The following decision tree is for the concept buy_computer that indicates whether a customer at a company is likely to buy a computer or not. Each internal node represents a test on an attribute. Each leaf node represents a class.



The benefits of having a decision tree are as follows −

- It does not require any domain knowledge.
- It is easy to comprehend.
- The learning and classification steps of a decision tree are simple and fast.

4 Baye's Theorem

Bayes' Theorem is named after Thomas Bayes. There are two types of probabilities −

- Posterior Probability [P(H/X)]
- Prior Probability [P(H)]

where X is data tuple and H is some hypothesis.

According to Bayes' Theorem,

$$P(H/X)= P(X/H)P(H) / P(X)$$

5 Bayesian Belief Network

Bayesian Belief Networks specify joint conditional probability distributions. They are also known as Belief Networks, Bayesian Networks, or Probabilistic Networks.

- A Belief Network allows class conditional independencies to be defined between subsets of variables.
- It provides a graphical model of causal relationship on which learning can be performed.
- We can use a trained Bayesian Network for classification.

There are two components that define a Bayesian Belief Network −

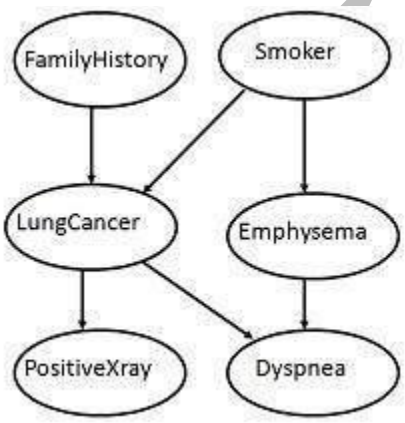
- Directed acyclic graph
- A set of conditional probability tables

6 Directed Acyclic Graph

- Each node in a directed acyclic graph represents a random variable.
- These variable may be discrete or continuous valued.
- These variables may correspond to the actual attribute given in the data.

7 Directed Acyclic Graph Representation

The following diagram shows a directed acyclic graph for six Boolean variables.



The arc in the diagram allows representation of causal knowledge. For example, lung cancer is influenced by a person's family history of lung cancer, as well as whether or not the person is a smoker. It is worth noting that the variable PositiveXray is independent of whether the patient has a family history of lung cancer or that the patient is a smoker, given that we know the patient has lung cancer.

8 IF-THEN Rules

Rule-based classifier makes use of a set of IF-THEN rules for classification. We can express a rule in the following from −

IF condition THEN conclusion

Let us consider a rule R1,

```
R1: IF age = youth AND student = yes
   THEN buy_computer= yes
```

Features −

- The IF part of the rule is called **rule antecedent** or **precondition**.
- The THEN part of the rule is called **rule consequent**.
- The antecedent part the condition consist of one or more attribute tests and these tests are logically ANDed.
- The consequent part consists of class prediction.

Note − We can also write rule R1 as follows −

```
R1: (age = youth) ^ (student = yes))(buys computer = yes)
```

If the condition holds true for a given tuple, then the antecedent is satisfied.

9 Rule Extraction

Here we will learn how to build a rule-based classifier by extracting IF-THEN rules from a decision tree.

Features −

To extract a rule from a decision tree −

- One rule is created for each path from the root to the leaf node.
- To form a rule antecedent, each splitting criterion is logically ANDed.
- The leaf node holds the class prediction, forming the rule consequent.

10 Clustering

Clustering is the process of making a group of abstract objects into classes of similar objects.

- A cluster of data objects can be treated as one group.
- While doing cluster analysis, we first partition the set of data into groups based on data similarity and then assign the labels to the groups.
- The main advantage of clustering over classification is that, it is adaptable to changes and helps single out useful features that distinguish different groups.

Applications of Cluster Analysis

- Clustering analysis is broadly used in many applications such as market research, pattern recognition, data analysis, and image processing.
- Clustering can also help marketers discover distinct groups in their customer base. And they can characterize their customer groups based on the purchasing patterns.
- In the field of biology, it can be used to derive plant and animal taxonomies, categorize genes with similar functionalities and gain insight into structures inherent to populations.
- Clustering also helps in identification of areas of similar land use in an earth observation database. It also helps in the identification of groups of houses in a city according to house type, value, and geographic location.
- Clustering also helps in classifying documents on the web for information discovery.
- Clustering is also used in outlier detection applications such as detection of credit card fraud.
- As a data mining function, cluster analysis serves as a tool to gain insight into the distribution of data to observe characteristics of each cluster.

11. KNN Algorithm

KNN can be used for both classification and regression predictive problems. However, it is more widely used in classification problems in the industry. To evaluate any technique we generally look at 3 important aspects:

1. Ease to interpret output

2. Calculation time

3. Predictive Power

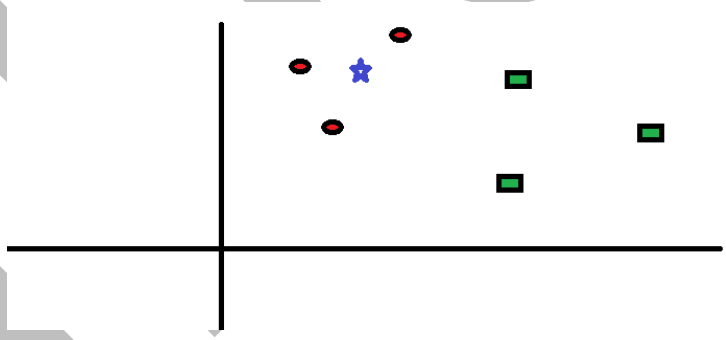
Let us take a few examples to place KNN in the scale :

	Logistic Regression	CART	Random Forest	KNN
1. Ease to interpret output	2	3	1	3
2. Calculation time	3	2	1	3
3. Predictive Power	2	2	3	2

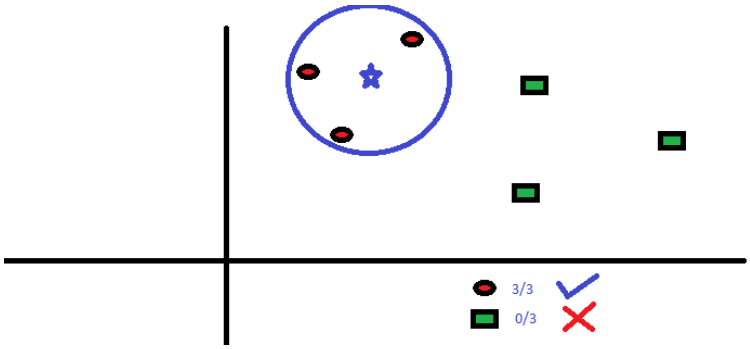
KNN algorithm fairs across all parameters of considerations. It is commonly used for its easy of interpretation and low calculation time.

How does the KNN algorithm work?

Let's take a simple case to understand this algorithm. Following is a spread of red circles (RC) and green squares (GS) :



You intend to find out the class of the blue star (BS). BS can either be RC or GS and nothing else. The "K" is KNN algorithm is the nearest neighbors we wish to take vote from. Let's say K = 3. Hence, we will now make a circle with BS as center just as big as to enclose only three datapoints on the plane. Refer to following diagram for more details:



The three closest points to BS is all RC. Hence, with good confidence level we can say that the BS should belong to the class RC. Here, the choice became very obvious as all three votes from the closest neighbor went to RC. The choice of the parameter K is very crucial in this algorithm. Next we will understand what are the factors to be considered to conclude the best K.

KNN algorithm:

```
Input:
   D      //Training data
   K      //Number of neighbors
   t      //Input tuple to classify
Output:
   c      //Class to which t is assigned
KNN Algorithm:
          //Algorithm to classify tuple using KNN
   N = ∅;
                //Find set of neighbors, N, for t
   foreach d ∈ D do
      if | N |≤ K then
         N = N ∪ d;
      else
         if ∃ u ∈ N such that sim(t, u) ≥ sim(t, d) then
            begin
               N = N − u;
               N = N ∪ d;
            end
            //Find class for classification
   c = class to which the most u ∈ N are classified;
```

11 Association rules

They are if-then statements that help to show the probability of relationships between data items within large data sets in various types of databases. Association rule mining has a number of applications and is widely used to help discover sales correlations in transactional data or in medical data sets.

How association rules work

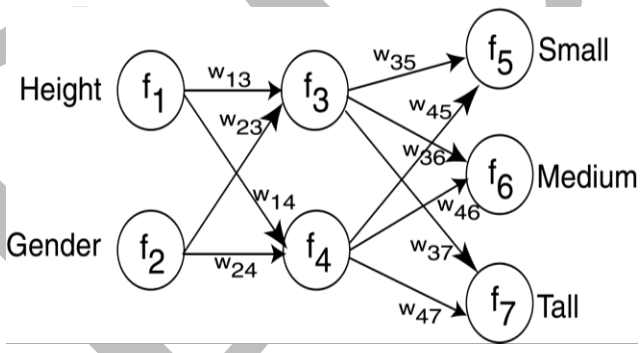
Association rule mining, at a basic level, involves the use of machine learning models to analyze data for patterns, or co-occurrence, in a database. It identifies frequent if-then associations, which are called *association rules*.

An association rule has two parts: an antecedent (if) and a consequent (then). An antecedent is an item found within the data. A consequent is an item found in combination with the antecedent.

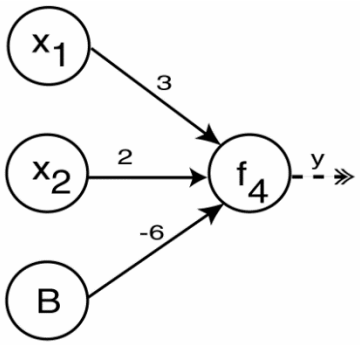
Association rules are created by searching data for frequent if-then patterns and using the criteria *support* and *confidence* to identify the most important relationships. Support is an indication of how frequently the items appear in the data. Confidence indicates the number of times the if-then statements are found true. A third metric, called *lift*, can be used to compare confidence with expected confidence.

Association rules are calculated from *itemsets*, which are made up of two or more items. If rules are built from analyzing all the possible itemsets, there could be so many rules that the rules hold little meaning. With that, association rules are typically created from rules well-represented in data.

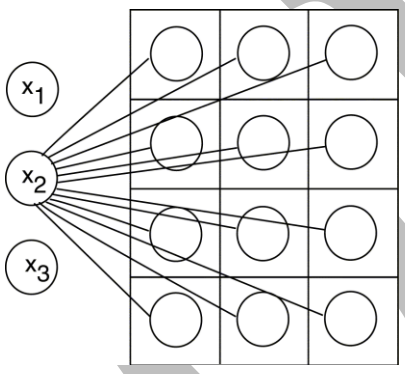
12 Neural Networks



- Mainly 3 models.
  - Perceptrons
    - Perceptron is one of the simplest Neural Network.
    - No hidden layers.



- Back propagation networks
- Kohonen Self Organizing Maps
    - Brain has different places called visual maps, maps of spatial possibilities etc.
    - Initially, SOM has a random assignment of vectors to each unit.
    - During training, these vectors are incrementally adjusted to give better coverage of the same.
    - Competitive Unsupervised Learning.
    - Observe how neurons work in brain:
        - Firing impacts firing of those near.
        - Neurons far apart inhibit each other.
        - Neurons have specific non-overlapping tasks.



13 Genetic Algorithms

The idea of genetic algorithm is derived from natural evolution. In genetic algorithm, first of all, the initial population is created. This initial population consists of randomly generated rules. We can represent each rule by a string of bits.

For example, in a given training set, the samples are described by two Boolean attributes such as A1 and A2. And this given training set contains two classes such as C1 and C2.

We can encode the rule **IF A1 AND NOT A2 THEN C2** into a bit string **100**. In this bit representation, the two leftmost bits represent the attribute A1 and A2, respectively.

Likewise, the rule **IF NOT A1 AND NOT A2 THEN C1** can be encoded as **001**.

Note − If the attribute has K values where K>2, then we can use the K bits to encode the attribute values. The classes are also encoded in the same manner.

- Based on the notion of the survival of the fittest, a new population is formed that consists of the fittest rules in the current population and offspring values of these rules as well.
- The fitness of a rule is assessed by its classification accuracy on a set of training samples.
- The genetic operators such as crossover and mutation are applied to create offspring.
- In crossover, the substring from pair of rules are swapped to form a new pair of rules.
- In mutation, randomly selected bits in a rule's string are inverted.

16.7 10 Rules: Setting a Good Data Mining Environment

- Support extremely large data set.
- Support hybrid learning: classification …
- Establish a Data Warehouse.
- Introduce data cleaning facilities.
- Facilitate working with dynamic coding.
- Integrate with DSS.
- Choose extendable architecture.
- Support heterogeneous Databases.
- Introduce Client-Server architecture.
- Introduce cache optimization.

Exercise:

1. What is data mining? State its applications.
2. Explain supervised and unsupervised learning.
3. What is Knowledge Data Discovery?
4. Explain the stages of KDD process.
5. Give comparison between data mining and KDD.
6. Explain Association mining rule.
7. Explain Decision Tree algorithm.
8. Explain KNN algorithm.
9. Explain 10 golden rules for setting a good data mining environment