Words of Concern

Point to be As Allan Bloom has said "Education is the movement from darkness to light". Through this handbook, I have tried to illuminate what might otherwise appear as black boxes to some. In doing so, I have used references from several other authors to synthesize or simplify or elaborate information. This is not possible without omitting details that I deem trivial while dilating the data that I consider relevant to topic. Every effort has been made to avoid errors. In spite of this, some errors might have crept in. Any errors or discrepancies noted maybe brought to my notice which I shall rectify in my next revision.

This handbook is solely for educational purpose and is not for sale. This handbook shall not be reproduced or distributed or used for commercial purposes in any form or by any means.

Thanks,
Deeksha.V.
Interface College of Computer Applications (ICCA)
Davangere

Bibliography

1. Stuart Russel, Peter Norvig: Artificial Intelligence A Modern Approach, 2nd Edition, Pearson Education 2003
2. Tom Mitchell, "Machine Learning", 1st Edition, McGraw-Hill,2017.
3. Elaine Rich, Kevin Knight, Shivashankar B Nair: Artificial Intelligence, Tata McGraw Hill 3rd edition

Interface College of Computer Applications (ICCA)

Syllabus:

Course Code: DSC16
Course Title: Artificial Intelligence and Applications (Theory)
Credits - 04
Sem - VI
Hours- 52
Formative Assessment Marks: 40
Summative Assessment Marks: 60
Duration of SEA/Exam: 02 hrs.

Course Outcomes (COs): After the successful completion of the course, the student will be able to:

1. Gain a historical perspective of AI and its foundations.
2. Become familiar with basic principles and strategies of AI towards problem solving.
3. Understand and apply approaches of inference, perception, knowledge representation, and learning.
4. Understand the various applications of AI.

Contents **52 Hrs**

Unit I 10 Hrs

Introduction- What is Artificial Intelligence, Foundations o AI, History, AI-past, present and Future. Intelligent Agents- Environments- Specifying the task environment, Properties of task environments, Agents based programs-Structure of Agents , Types of agents- Simple reflex agents, Model-based reflex agents, Goal-based agents; and Utility-based agents.

Unit II 10 Hrs

Problem Solving by Searching-Problem-Solving Agents, Well-define problems and solutions, examples Problems, Searching for Solutions, Uninformed Search -Breadth-firs search, Uniform-cost search, Depth-first search, Depth-limited search, Iterative deepening depth-first search, Bidirectional search, Greedy best-first search, A* Search, AO* search Informed (Heuristic) Search Strategies, Heuristic Functions.

Unit III 12 Hrs

Knowledge Representation - Knowledge-Based Agents, The Wumpus World Logic, Propositional Logic, Propositional Theorem Proving, Effective Propositional Model Checking, Agents Based on Propositional Logic, First-Order Logic-Syntax and Semantics of First-Order Logic, Using First-Order Logic, Unification and Lifting Forward Chaining, Backward Chaining.

Interface College of Computer Applications (ICCA)

Unit IV 10 Hrs

Learning- Forms of Learning, Supervised Learning, Machine Learning Decision Trees, Regression and Classification with Linear Models, Artificial Neural Networks, Support Vector Machines.

Unit V 10 Hrs

Applications of AI - Natural Language Processing, Text Classification and Information Retrieval, Speech Recognition, Image processing and computer vision, Robotics.

Unit 1 : Introduction

Dartmouth Conference(1956, where AI research officially began.

What is Artificial Intelligence

- Artificial Intelligence is the branch of computer science concerned with making computers behave like humans.
- Major AI textbooks define artificial intelligence as "the study and design of intelligent
- agents," where an intelligent agent is a system that perceives its environment and takes actions which maximize its chances of success.
- John McCarthy, who coined the term in 1956, defines it as "the science and engineering of making intelligent machines, especially intelligent computer
- programs."
- The definitions of AI according to some text books are categorized into four approaches and are summarized in the table below :.

Systems that think like humans	Systems that think rationally
"The exciting new effort to make computers think . . . *machines with minds*, in the full and literal sense." (Haugeland, 1985) "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning . . ." (Bellman, 1978)	"The study of mental faculties through the use of computational models." (Chamiak and McDermott, 1985) "The study of the computations that make it possible to perceive, reason, and act." (Winston, 1992)
Systems that act like humans	**Systems that act rationally**
"The art of creating machines that perform functions that require intelligence when performed by people." (Kurzweil, 1990) "The study of how to make computers do things at which, at the moment, people are better." (Rich and Knight, 1991)	"Computational Intelligence is the study of the design of intelligent agents." (Poole *et al.*, 1998) "AI . . .is concerned with intelligent behavior in artifacts." (Nilsson, 1998)

Figure 1.1 Some definitions of artificial intelligence, organized into four categories.

The four main approaches or perspectives that define the goals and principles of AI systems.

Interface College of Computer Applications (ICCA)

1. Acting Humanly: The Turing Test
    - This approach focuses on developing systems that can mimic and replicate human behaviour, cognition, and decision-making processes.
    - The goal is to create AI systems that can perform tasks and solve problems in a way that is indistinguishable from how humans would perform them.

The Turing Test

- The Turing Test, proposed by Alan Turing in 1950, is a fundamental concept in the "Acting Humanly" approach to artificial intelligence.
- It is a test of a machine's ability to exhibit intelligent behaviour that is indistinguishable from a human.

The Turing Test involves the following setup:

- A human evaluator (interrogator) interacts with two entities through a text-based interface, where the identities of the entities are hidden.

- One of the entities is a human, and the other is a computer program (the AI system being tested).
- The evaluator can ask any questions or engage in any conversational topic with the two entities through messages.
- If the evaluator cannot reliably distinguish the AI system from the human based on their responses, the AI system is considered to have passed the Turing Test.

To pass the Turing Test, an AI system would need to demonstrate the following abilities:

- **Natural language** processing to enable it to communicate successfully in English.
- **Knowledge representation** to store what it knows or hears;
- **automated reasoning** to use the stored information to answer questions and to draw new conclusions;
- **machine learning** to adapt to new circumstances and to detect and extrapolate patterns.

2. Think Humanly: The cognitive approach

- This approach aims to develop AI systems that can think and reason like humans, by trying to understand and model the underlying cognitive processes and mental mechanisms of the human mind.
- It involves studying and replicating the processes of human reasoning, problem-solving, decision-making, and learning.

The cognitive approach

- The cognitive approach in Artificial Intelligence (AI) aims to create machines that think like humans.
- It focuses on understanding how the human mind processes information, solves problems, and learns, and then replicating those processes in computer programs
- Researchers in cognitive science, a field that combines psychology, neuroscience, and computer science, use various methods to achieve this:
    - Introspection: People try to analyze their own thought processes.
    - Psychological Experiments: Controlled experiments observe human behavior in response to stimuli to understand decision-making and problem-solving patterns.
    - Brain Imaging: Techniques like fMRI scans provide insights into brain activity during different cognitive tasks.

Interface College of Computer Applications (ICCA)

3. Thinking Rationally

- The "laws of thought" approach, also known as "Thinking Rationally," is another perspective in artificial intelligence (AI) that aims to develop AI systems that can reason logically and draw correct conclusions based on available information and formal rules of reasoning.
- Aristotle was one of the first to attempt to codify —"right thinking", that is, irrefutable reasoning processes.
- His syllogisms provided patterns for argument structures that always yielded correct conclusions when given correct premises.
- Eg. Socrates is a man; all men are mortal; therefore, Socrates is mortal.-- logic
- There are two main obstacles to this approach.
    1. it is not easy to take informal knowledge and state it in the formal terms required by logical notation, particularly when the knowledge is less than 100% certain.
    2. Second, there is a big difference between solving a problem —in principle and solving it in practice.

4. Acting Rationally: The rational agent approach
    - An agent is something that acts.
    - Computer agents are not mere programs ,but they are expected to have the following attributes also :
      (a) operating under autonomous control
      (b) perceiving their environment
      (c) persisting over a prolonged time period
      (e) adapting to change.
    - A rational agent is one that acts so as to achieve the best outcome.

Foundation of AI

The various disciplines that contributed ideas, viewpoints, and techniques to AI are given below :

1. Philosophy(428 B.C. — present)
    - ➢ Where does knowledge come from?
    - ➢ How does knowledge lead to action?
    - Aristotle (384—322 B.C.): was the first to formulate a precise set of laws governing the rational part of the mind.
    - Thomas Hobbes (1588—1679) proposed that reasoning was like numerical computation, that "we add and subtract in our silent thoughts."
    - Rene Descartes(1596-1650): Developed dualistic theory of mind and matter. Descartes attempted to demonstrate the existence of god and the distinction between the human soul and body.
    - The empiricism movement, starting with Francis Bacon's (1561—1626).
    - The confirmation theory of Carnap and Carl Hempel (1905—1997) attempted to analyze the acquisition of knowledge from experience.
    - Carnap's book The Logical Structure of the World (1928) defined an explicit computational procedure for extracting knowledge from elementary experiences. It was probably the first theory of mind as a computational process.
    - The final element in the philosophical picture of the mind is the connection between
    - knowledge and action. This question is vital to AI because intelligence requires action as well as reasoning.

2. Mathematics
    - ➢ What are the formal rules to draw valid conclusions?
    - ➢ What can be computed?
    - George Boole (1815—1864), who worked out the details of propositional, or Boolean, logic (Boole, 1847).
    - In 1879, Gottlob Frege (1848—1925) extended Boole's logic to include objects and relations, creating the first
    - order logic that is used today.
    - The first nontrivial algorithm is thought to be Euclid's algorithm for computing greatest common divisors.
    - Besides logic and computation, the third great contribution of mathematics to AI is the PROBABILITY theory of probability. The Italian Gerolamo Cardano (1501—1576) first framed the idea of probability, describing it in terms of the possible outcomes of gambling events.
    - Thomas Bayes (1702—1761) proposed a rule for updating probabilities in the light of new evidence. Bayes' rule underlies most modern approaches to uncertain reasoning in AI systems.

Interface College of Computer Applications (ICCA)

3. Economics
   - ➢ How should we make decisions to maximize payoff?
   - ➢ How should we do this when the payoff may be far in the future?
   - The science of economics got its start in 1776, Smith was the first to treat it as a
   - science, using the idea that economies can be thought of as consisting of individual agents maximizing their own economic well-being.
   - Decision theory, which combines probability theory with utility theory, provides a formal and complete framework for decisions (economic or otherwise) made under uncertainty.
   - Von Neumann and Morgenstern's development of game theory included the surprising result that, for some games, a rational agent should adopt policies that are (or least appear to be) randomized. Unlike decision theory, game theory does not offer an unambiguous prescription for selecting actions.

4. Neuroscience
   - ➢ How do brains process information?
   - Neuroscience is the study of the nervous system, particularly the brain.
   - 335 B.C. Aristotle wrote, "Of all the animals, man has the largest brain in
   - proportion to his size."
   - Nicolas Rashevsky (1936, 1938) was the first to apply mathematical models to
   - the study of the nervous system.
   - The measurement of intact brain activity began in 1929 with the invention by Hans Berger of the electroencephalograph (EEG).
   - The recent development of functional magnetic resonance imaging (fMRI) (Ogawa et al., 1990; Cabeza and Nyberg, 2001) is giving neuroscientists unprecedentedly detailed images of brain activity, enabling measurements that correspond in interesting ways to ongoing cognitive processes.

5. Psychology
   - ➢ How do humans and animals think and act?
   - Behaviourism movement, led by John Watson(1878-1958). Behaviorists insisted on studying only objective measures of the percepts(stimulus) given to an animal and its resulting actions(or response). Behaviorism discovered a lot about rats and pigeons but had less success at understanding human.
   - Cognitive psychology , views the brain as an information processing device. Common view among psychologist that a cognitive theory should be like a computer program.(Anderson 1980) i.e. It should describe a detailed information processing mechanism whereby some cognitive function might be implemented.

ICCA
Interface College of Computer Applications (ICCA)

6. Computer Science
   - ➢ How can we build an efficient computer?
   - The first operational computer was the electromechanical Heath Robinson,8 built in 1940 by Alan Turing's team for a single purpose: deciphering German messages.
   - The first operational programmable computer was the Z-3, the invention of Konrad Zuse in Germany in 1941.
   - The first electronic computer, the ABC, was assembled by John tanasoff and his student Clifford Berry between 1940 and 1942 at Iowa State University.
   - The first programmable machine was a loom, devised in 1805 by Joseph Marie Jacquard (1752—1834), that used punched cards to store instructions for the pattern to be woven.

7. Control theory and Cybernetics

- ➢ How can artifacts operate under their own control?
- Ktesibios of Alexandria (c. 250 B.C.) built the first self-controlling machine: a water clock with a regulator that maintained a constant flow rate. This invention changed the definition of what an artifact could do.
- Modern control theory, especially the branch known as stochastic optimal control, has as its goal the design of systems that maximize an objective function over time. This roughly matches our view of AI: designing systems that behave optimally.

8. Linguistics
   - ➢ How does language relate to thought?
   - In 1957, B. F. Skinner published Verbal Behavior. This was a comprehensive, detailed account of the behaviorist approach to language learning, written by the foremost expert in the field.
   - Noam Chomsky, who had just published a book on his own theory, Syntactic Structures. Chomsky pointed out that the behaviorist theory did not address the notion of creativity in language.
   - Modern linguistics and AI, then, were "born" at about the same time, and grew up together, intersecting in a hybrid field called computational linguistics or natural language processing.

History of AI

1. 1950s: The Birth of AI
   - 1950: Alan Turing's "Computing Machinery and Intelligence" paper considered the possibility of building an intelligent machine.
   - 1956: The term "Artificial Intelligence" was coined at the Dartmouth Conference organized by John McCarthy.
   - Early AI researchers were optimistic about creating intelligent systems like general problem solvers, geometry provers, and language understanding programs.
2. 1960s: The First AI Winter
   - Progress was slower than expected, and initial optimism faded.
   - AI systems were limited to solving "toy" problems within simplified domains.
   - Funding from governments and institutions decreased, leading to the first "AI winter."
3. 1970s: Knowledge-Based Systems
   - The development of the programming language LISP facilitated symbolic reasoning techniques.
   - Knowledge-based systems employed human-like expertise and rules for problem-solving in specific domains.
   - Examples include DENDRAL (for analyzing chemical compounds) and MYCIN (for medical diagnosis).
4. 1980s: The AI Revival
   - The success of expert systems in limited domains renewed interest and funding in AI.
   - New parallel processing hardware offered the prospect of faster computation.
   - However, the limitations of expert systems became evident, leading to a search for new approaches.
5. 1990s: AI Becomes an Industry
   - Machine learning techniques, particularly artificial neural networks, gained popularity.
   - AI systems were deployed in areas like speech recognition, data mining, and robotics.

Interface College of Computer Applications (ICCA)

- The advent of the internet and the availability of large datasets fueled the growth of AI applications.

6. 2000s: The Dawn of Modern AI
   - Significant progress was made in machine learning, particularly with the rise of deep learning techniques.
   - AI systems surpassed human performance in various tasks, such as object recognition, game-playing, and natural language processing.
   - Tech giants like Google, Microsoft, and Facebook invested heavily in AI research and development.

AI – Past, Present and Future

Past:

- Early Foundations (1950s-1960s):
  - The term "artificial intelligence" was coined in 1956 during the Dartmouth Conference.
  - Early AI research focused on symbolic reasoning, problem-solving, and logical inference.
  - Significant developments include the Logic Theorist by Newell and Simon and the General Problem Solver.
- AI Winter (1970s-1980s):
  - Despite initial enthusiasm, progress in AI slowed down due to high expectations and limited computing power.
  - Funding cuts and disillusionment with AI's progress led to a period known as the "AI winter."
- Resurgence (1990s-2000s):
  - Advances in machine learning, neural networks, and computational power sparked a resurgence of interest in AI.
  - The development of expert systems, natural language processing, and early versions of virtual assistants emerged during this time.

Present:

- Deep Learning Revolution (2010s-Present):
  - Breakthroughs in deep learning, fuelled by large datasets and powerful GPUs, have revolutionized AI.
  - Deep neural networks have achieved remarkable success in image recognition, natural language processing, and speech recognition.
  - Applications of AI permeate various industries, including healthcare, finance, transportation, and entertainment.
- Narrow AI Dominance:
  - Current AI systems excel in specific tasks but lack general intelligence.
  - Narrow AI applications include recommendation systems, autonomous vehicles, virtual assistants, and fraud detection.
- Ethical and Societal Concerns:
  - As AI becomes more pervasive, concerns about privacy, bias, job displacement, and autonomous decision-making have grown.
  - Efforts to develop ethical guidelines, regulation, and responsible AI practices are underway.

Future:

- Advancements in General AI:
  - Researchers aim to develop artificial general intelligence (AGI) capable of reasoning, learning, and adapting across various domains.
  - AGI could potentially surpass human intelligence and lead to transformative societal changes.
- Human-AI Collaboration:

Interface College of Computer Applications (ICCA)

- AI systems will increasingly work alongside humans, augmenting human capabilities in various fields.
- Collaboration between humans and AI could lead to improved productivity, creativity, and problem-solving.

- Addressing Ethical and Societal Challenges:
  - Future AI development will require addressing ethical dilemmas, ensuring transparency, fairness, and accountability.
  - Efforts to mitigate biases, protect privacy, and establish regulatory frameworks will be crucial for the responsible deployment of AI.
- AI in Exploration and Innovation:
  - AI will play a significant role in scientific discovery, exploration, and innovation, aiding research in fields such as medicine, climate science, and space exploration.

Intelligent Agents

1. Agents

- An agent can be anything that perceive its environment through sensors and act upon that environment through actuators.
- An agent can be:
  - Human-Agent: A human agent has eyes, ears, and other organs which work for sensors and hand, legs, vocal tract work for actuators.
  - Robotic Agent: A robotic agent can have cameras, infrared range finder, NLP for sensors and various motors for actuators.
  - Software Agent: Software agent can have keystrokes, file contents as sensory input and act on those inputs and display output on the screen.

2. Intelligent agents

- An intelligent agent is an autonomous entity that perceives its environment through sensors and acts upon that environment through actuators to achieve specific goals.
- Following are the main four rules for an AI agent:
  - Rule 1: An AI agent must have the ability to perceive the environment.
  - Rule 2: The observation must be used to make decisions.
  - Rule 3: Decision should result in an action.
  - Rule 4: The action taken by an AI agent must be a rational action.

Interface College of Computer Applications (ICCA)

3. Rational agents

- These are intelligent agents that behave rationally like human beings, and their actions are based on perceived sequence and logical reasoning.
- They can be machines, software programs, or actuating programmable devices or mechanisms.
- Their characteristics can be summarized as listed below.
  - Ability to make correct decisions based on previous experience or information and logical thinking.
  - Ability to take action on clear and sound preferences apt to existing environmental status.
  - A clear understanding of the effect on the environment by the performed actions.
  - Ability to monitor and act promptly, even in a few uncertain situations.
  - Well-informed about the rewards and penalties based on successes or failures and hence the ability to learn for improvements, if needed.

Agents and Environment

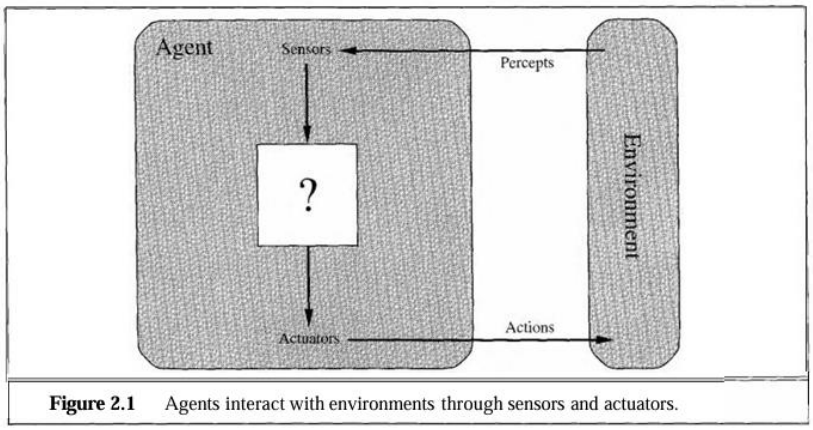


Figure 2.1 Agents interact with environments through sensors and actuators.

- Sensor: Sensor is a device which detects the change in the environment and sends the information to other electronic devices. An agent observes its environment through sensors.
- Actuators: Actuators are the component of machines that converts energy into motion. The actuators are only responsible for moving and controlling a system. An actuator can be an electric motor, gears, rails, etc.
- Effectors: Effectors are the devices which affect the environment. Effectors can be legs, wheels, arms, fingers, wings, fins, and display screen
- Percepts and Percept Sequences: Percepts refer to the inputs or sensory information that an agent receives from its environment at any given moment. A percept sequence is the complete history of all percepts the agent has ever perceived.
- Action: Based on its perception, the agent can take actions in the environment. These actions can be physical (e.g., moving a robot arm) or virtual (e.g., sending commands to a software system).
- Agent function: An agent's behavior is described by the agent function that maps any given percept sequence to an action.
  f : P* -> A
- Agent program: Internally, The agent function for an artificial agent will be implemented by an agent program.

Interface College of Computer Applications (ICCA)

Note:- The agent function is an abstract mathematical description; the agent program is a concrete implementation, running on the agent architecture.

Specifying the task environment (PEAS Description)

- The task environment in artificial intelligence (AI) refers to the problem domain or environment in which an intelligent agent operates and performs its tasks.
- It defines the conditions, constraints, and objectives that the agent must consider and address while interacting with the environment to achieve its goals.
- The task environment is a crucial aspect of designing and developing intelligent agents because it provides the context and boundaries within which the agent must perceive, reason, and act.
- It serves as a blueprint for tailoring the agent's capabilities, decision-making strategies, and behaviors to the specific problem at hand.
- When specifying the task environment in AI, several key aspects are typically considered:
  - Performance Measure: This specifies the criteria or metrics by which the agent's success or failure will be evaluated. It defines the desired behavior or goals that the agent should try to achieve or maximize.

- Environment: This describes the environment or problem domain in which the agent operates. It includes characteristics such as whether the environment is fully observable or partially observable, deterministic or stochastic, episodic or sequential, static or dynamic, discrete or continuous, and single-agent or multi-agent.
- Actuators: These are the available means or effectors by which the agent can take actions and influence the environment. Actuators can be physical (e.g., robotic arms, motors) or software-based (e.g., sending commands, modifying data).
- Sensors: These are the mechanisms or inputs through which the agent can perceive and observe the state of the environment. Sensors can be physical (e.g., cameras, microphones, touch sensors) or virtual (e.g., reading data from files or databases).

Example

Agent Type	Performance Measure	Environment	Actuators	Sensors
Taxi driver	Safe: fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers	Steering, accelerator, brake, signal, horn, display	Cameras, sonar, speedometer, GPS, odometer, accelerometer, engine sensors, keyboard

Figure 2.4 PEAS description of the task environment for an automated taxi.

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, minimize costs, lawsuits	Patient, hospital, staff	Display questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display categorization of scene	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, joint angle sensors
Refinery controller	Maximize purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, chemical sensors
Interactive English tutor	Maximize student's score on test	Set of students, testing agency	Display exercises, suggestions, corrections	Keyboard entry

Figure 2.5 Examples of agent types and their PEAS descriptions.

(ICCA)

Properties of Task Environment

- The task environment in artificial intelligence (AI) can be characterized by various properties, which influence the design and development of intelligent agents.
- These properties help to define the challenges and constraints that the agent must address while operating in the environment.
- The main properties of the task environment are:

1. Observable vs. Partially Observable:
    - Observable: The agent has complete access to the environment's state and can perceive all relevant information.
    - Example: A chessboard is fully observable for a chess-playing AI)
    - Partially Observable: The agent can only perceive a partial view of the environment's state, and some information is hidden or unavailable.
    - Example: A self-driving car encountering hidden obstacles due to fog)

2. Deterministic vs. Stochastic vs Strategic:
    - Deterministic: The next state of the environment is entirely determined by the current state and the agent's actions, with no uncertainty or randomness involved.
      Example: - Solving Mathematical Equations: Solving a set of mathematical equations is a deterministic process. Given the same input values and operations, the solution will always be the same.
    - Stochastic: The environment's next state is influenced by random events or probabilistic processes, introducing uncertainty and unpredictability.
      Example: - A dice game, where the outcome of each roll is probabilistic and introduces uncertainty in the environment.
    - Strategic: - A strategic environment is a type of environment in which an agent or decision-maker needs to consider not only the current state of the environment but also the potential actions and strategies of other agents or adversaries present in the environment.
      Example:- Chess, where a person need to think about the pan movement of the opponent.

3. Episodic vs. Sequential:
    - In an Episodic task environment, each of the agent's actions is divided into atomic incidents or episodes. There is no dependency between current and previous incidents. In each incident, an agent receives input from the environment and then performs the corresponding action.
      Example: Consider an example of Pick and Place robot, which is used to detect defective parts from the conveyor belts. Here, every time robot(agent) will make the decision on the current part i.e. there is no dependency between current and previous decisions.
    - Sequential: In sequential environments, decisions made at one point can impact future outcomes. Actions taken now can have consequences that affect subsequent decisions.
      Example: Chess, where the previous move can affect all the following moves.

4. Static vs. Dynamic:
    - Static: The environment remains fixed and does not change while the agent is deliberating or acting.
      Example: - A game of chess is played on a static board with fixed rules. The board configuration and the pieces' positions change, but the environment itself remains static throughout the game.

Interface College of Computer Applications (ICCA)

- Dynamic: The environment can change while the agent is deliberating or acting, independently of the agent's actions.
  Example: - Robot Navigation: When a robot navigates through an environment, it encounters a dynamic environment where obstacles, other agents, or environmental conditions can change over time.

5. Discrete vs. Continuous:
   - Discrete: The environment's state, time, and action spaces are discrete, with a finite number of possible values.
     Example: - A grid-based game environment, where the agent's position and actions are discrete
   - Continuous: The environment's state, time, and action spaces are continuous, with an infinite number of possible values.
     Example: - A robot manipulator controlling a robotic arm, where the arm's positions and movements are continuous.

6. Single-agent vs. Multi-agent:
   - Single-agent: There is only one agent operating in the environment.
     Example: - A game of solitaire, where there is only one agent playing against the environment
   - Multi-agent: There are multiple agents operating in the environment, which may cooperate or compete with each other.
     Example: - A real-time strategy game, where multiple agents (players or AI-controlled units) interact and compete with each other.

7. Known vs. Unknown:
   - Known: The agent has complete knowledge of the environment's dynamics, including the state transition function and the effects of its actions.
     Example: - A simulated physics environment, where the agent has complete knowledge of the laws of motion and the effects of its actions.
   - Unknown: The agent does not have complete knowledge of the environment's dynamics and must learn or infer them through experience.
     Example:- A recommendation system for a new product, where the agent needs to learn user preferences and behavior through experience.

8. Accessible vs. Inaccessible:
   - Accessible: The agent can access and modify the state of the environment through its actions.
     Example:- A game environment where the agent can directly modify the game state by taking actions.
   - Inaccessible: The agent cannot directly access or modify the state of the environment and can only perceive it.
     Example :- A weather forecasting system, where the agent can only observe and predict the weather conditions but cannot directly modify them.

Interface College of Computer Applications (ICCA)

Task Environment	Observable	Deterministic	Episodic	Static	Discrete	Agents
Crossword puzzle	Fully	Deterministic	Sequential	Static	Discrete	Single
Chess with a clock	Fully	Strategic	Sequential	Semi	Discrete	Multi
Poker	Partially	Stochastic	Sequential	Static	Discrete	Multi
Backgammon	Fully	Stochastic	Sequential	Static	Discrete	Multi
Taxi driving	Partially	Stochastic	Sequential	Dynamic	Continuous	Multi
Medical diagnosis	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Image-analysis	Fully	Deterministic	Episodic	Semi	Continuous	Single
Part-picking robot	Partially	Stochastic	Episodic	Dynamic	Continuous	Single
Refinery controller	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Interactive English tutor	Partially	Stochastic	Sequential	Dynamic	Discrete	Multi

Figure 2.6 Examples of task environments and their characteristics.

Structure of Agents

- An agent can be considered as the combination of its architecture and its program.
- Together, the agent architecture and the agent program form the complete agent system:

  Agent = Agent Architecture + Agent Program(includes agent function)

- The agent architecture refers to the overall structure and components of the agent, including the sensors, actuators, knowledge base, memory, and any other supporting components. It defines how the different parts of the agent are organized and how they interact with each other.
- The agent program, on the other hand, is the decision-making component that determines how the agent should act based on its current percepts and knowledge. It specifies the algorithms, rules, or models that the agent uses to map its percepts and internal state to actions.

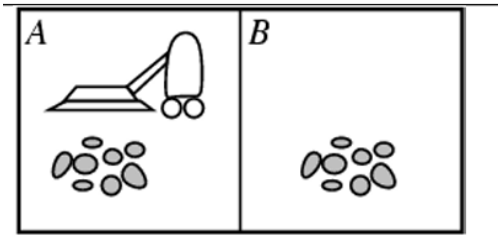
Agent based Programs

Interface College of Computer Applications (ICCA)

- **Agent function:-** An agent's behavior is described by the agent function that maps any given percept sequence to an action.

  f : P* -> A
- **Agent program:** Internally, The agent function for an artificial agent will be implemented by an agent program.

- To illustrate difference between agent function and agent program , we will use a very simple example-the vacuum-cleaner.
- This particular world has just two locations: squares A and B.



A vacuum cleaner world with just two locations.

- The vacuum agent perceives which square it is in and whether there is dirt in the square.
- It can choose to move left, move right, suck up the dirt, or do nothing.
- One very simple agent function is the following: if the current square is dirty, then suck, otherwise move to the other square.
- A partial tabulation of this agent function is shown in below figure.

Percept sequence	Action
[A, *Clean*]	*Right*
[A, *Dirty*]	Suck
[B,Clean]	*Left*
[B,Dirty]	Suck
[A,Clean],[A,Clean]	*Right*
[A,Clean],[A,Dirty]	Suck
[A,Clean],[A,Clean],[A,Clean]	*Right*
[A,Clean],[A,Clean],[A,Dirty]	Suck

Figure 2.3 Partial tabulation of a simple agent function for the vacuum-cleaner world shown in Figure 2.2.

- Agent program of this agent is shown in below figure .

```
function REFLEX-VACUUM-AGENT([location,status]) returns an action

  if status = Dirty then return Suck
  else if location = A then return Right
  else if location = B then return Left
```

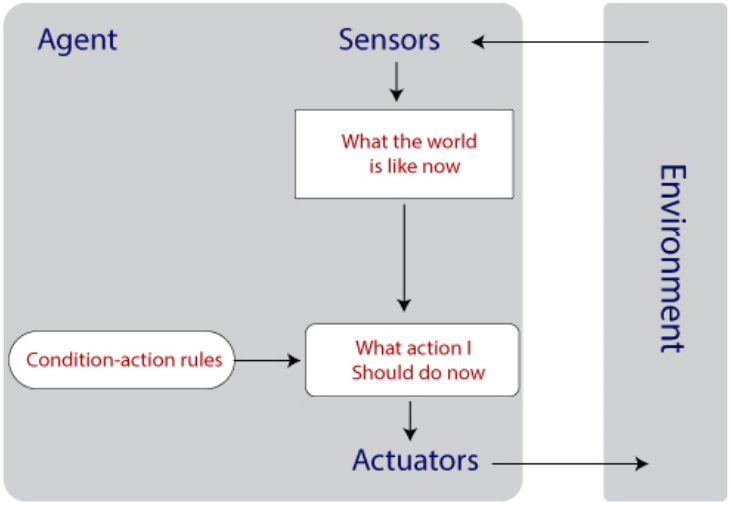
Figure 2.8 The agent program for a simple reflex agent in the two-state vacuum environment. This program implements the agent function tabulated in Figure 2.3.

Types of agents

Interface College of Computer Applications (ICCA)

1. Simple Reflex agents

- Simple reflex agents are one of the most basic types of agents in artificial intelligence (AI).
- They operate based on a simple condition-action rule or a set of rules, without considering the entire percept history or making decisions based on a complex internal model of the environment.
- A simple reflex agent follows the following principles:
  - Perception: The agent perceives the current state of the environment through its sensors.
  - Condition-Action Rules: The agent has a set of condition-action rules that map perceived conditions or situations to actions.
  - Action Selection: The agent selects and performs the action that corresponds to the perceived condition, according to the condition-action rules.



Schematic diagram of a simple-reflex agent

The decision-making process of a simple reflex agent can be represented as:

```
function SIMPLE-REFLEX-AGENT(percept) returns an action
    persistent: rules, a set of condition-action rules
    state <- INTERPRET-INPUT(percept)
    rule <- RULE-MATCH(state, rules)
    action <- rule.ACTION
    return action
```

Where,

1. function SIMPLE-REFLEX-AGENT(percept) returns an action: This line defines a function called SIMPLE-REFLEX-AGENT that takes a percept as input and returns an action.
2. persistent: rules, a set of condition-action rules: This line indicates that the function has a persistent set of condition-action rules that are used to determine the action to be taken.
3. state <- INTERPRET-INPUT(percept): This line calls a function called INTERPRET-INPUT that takes the percept as input and returns a state representation of the environment. The <- symbol is an assignment operator in pseudocode.
4. rule <- RULE-MATCH(state, rules): This line calls a function called RULE-MATCH that takes the state and the set of rules as input. It matches the current state against the condition-action rules and returns the matching rule.
5. action <- rule.ACTION: This line extracts the action part of the matched rule and assigns it to the action variable.
6. return action: This line returns the selected action as the output of the SIMPLE-REFLEX-AGENT function.

For example, a simple reflex agent for a robot vacuum cleaner might have rules like:

- If the vacuum detects dirt, turn on the suction motor.
- If the vacuum bumps into an obstacle, change direction.
- If the vacuum's battery is low, return to the charging station.

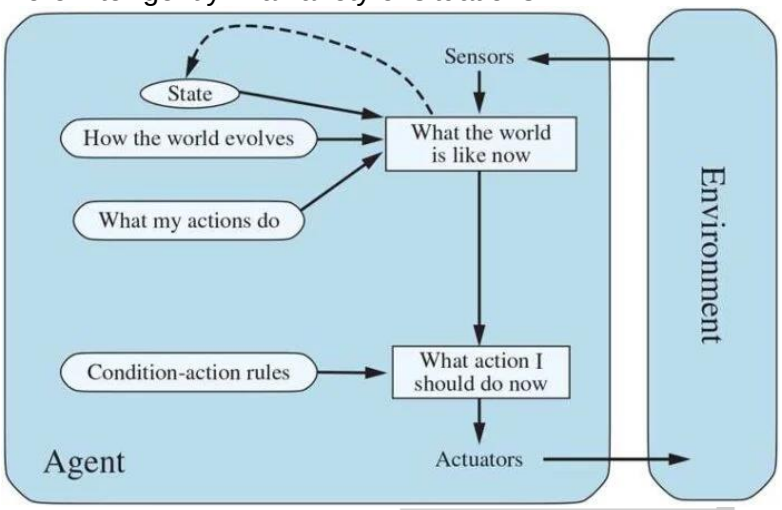
2. **Model – based reflex agents**
   - A model-based reflex agent is an agent that maintains an internal model of the environment and uses it to reason about the effects of its actions before executing them.

Interface College of Computer Applications (ICCA)

Working of Model based reflex agent

1. Percept: The agent receives input from the environment via sensors.
2. Update Internal State: It updates its internal state by considering the new percept and its previous state. This internal state is the agent's understanding of the world, which may include unseen aspects.
3. Condition-Action Rules: The agent has rules that connect the internal state to specific actions.
4. Action Selection: Based on its current internal state, the agent picks an action using these rules.
5. Execute Action: Finally, the agent performs the chosen action, influencing the environment.

In essence, a model-based reflex agent uses a representation of the world (a model) and its history of interactions (internal state) to make decisions about what actions to take, rather than relying solely on the current percept. This allows the agent to act more intelligently in a variety of situations.



A model- based reflex agent

The basic structure of a model-based reflex agent can be represented as follows:

Interface College of Computer Applications (ICCA)

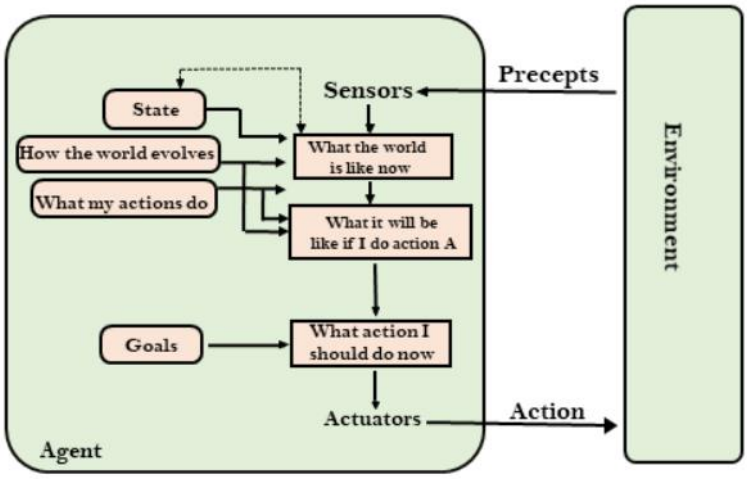
```
function MODEL-BASED-REFLEX-AGENT(percept) returns an action
   persistent: model, a representation of the current state
               rules, a set of condition-action rules
               action, the most recent action, initially none

   #   update model based on percept and action
   state <- UPDATE-STATE(model, action, percept)
   rule <- RULE-MATCH(state, rules)
   action <- rule.ACTION
   return action
```

Where,

1. Percept: The agent receives percepts from the environment, which provide information about the current state of the environment.
2. Model: The agent maintains an internal model of the environment, which represents the current state of the environment based on the percepts and the agent's previous actions.

3. Rules: The agent has a set of condition-action rules that determine how it should act in different situations.
4. UPDATE-STATE: This function updates the internal model of the environment based on the previous state, the previous action taken by the agent, and the current percept.
5. RULE-MATCH: This function matches the current state of the environment (as represented in the model) against the condition-action rules to determine which action the agent should take.
6. Action: The agent selects and performs the action dictated by the matched rule.

3. **Goal – based agents**
    - A goal-based AI agent refers to an artificial intelligence system designed to achieve specific objectives or goals within a given environment.
    - These agents are commonly used in various fields such as robotics, gaming, optimization, and automation

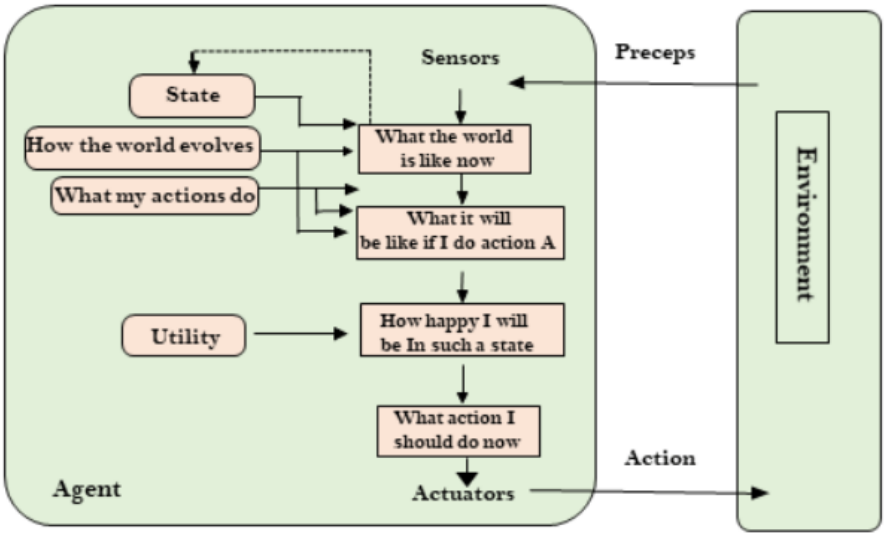


Core Components:

- Knowledge Base - Represents information about the environment, goals, constraints, and possible actions.
- Goal Representation - An explicit representation of the goal(s) the agent is trying to achieve (e.g., target state, reward function).
- Perception - Sensors to perceive the current state of the environment.
- Action Selection Mechanism - Uses techniques like planning, search, or reinforcement learning to choose which action to take next in pursuit of the goal.
- Action Execution - The ability to carry out actions that change the state of the environment towards the goal.
- Learning (optional) - Some agents can learn and update their knowledge/strategies based on experience.

Interface College of Computer Applications (ICCA)

4. **Utility – based agents**
    - Similar to goal-based agents but with an extra component: utility measurement.
    - These agents not only consider goals but also evaluate the best way to achieve them.
    - Utility functions map each state to a real number, indicating how efficiently an action achieves the goals.
    - Useful when there are multiple alternatives, and the agent must choose the most effective action.



Key components of Utility Based agent

1. Utility Function: This is a key component of a utility-based agent. It is a mathematical function that assigns a real number to each state of the environment, representing the degree of happiness, satisfaction, or preference of the agent for that state. The higher the number, the better the state is according to the agent's criteria.

2. Decision Making: Utility-based agents make decisions based on the expected utility of actions. They predict the outcome of actions and choose the one that is expected to lead to the highest utility. This allows them to handle situations with multiple possible alternatives and select the one that best aligns with their objectives.

Interface College of Computer Applications (ICCA)

Unit II :- Problem Solving by Searching

Problem Solving Agents

- Problem Solving agents in Artificial Intelligence, which are sort of goal-based agents. Because the straight mapping from states to actions of a basic reflex agent is too vast to retain for a complex environment, we utilize goal-based agents that may consider future actions and the desirability of outcomes.
- Problem Solving Agents decide what to do by finding a sequence of actions that leads to a desirable state or solution.
- An agent may need to plan when the best course of action is not immediately visible. They may need to think through a series of moves that will lead them to their goal state.
- Such an agent is known as a problem-solving agent, and the computation it does is known as a search.
- Problem-solving agents are a fundamental concept in artificial intelligence (AI). These agents are designed to tackle specific tasks or problems by generating sequences of actions to achieve desired outcomes.

The problem-solving agent follows this four-phase problem solving process:

1. Goal Formulation:
   - This is the first step where the agent identifies the desired end state or goal that needs to be achieved.
   - The goal formulation phase involves understanding the problem statement, constraints, and objectives.
   - AI agents can be designed to automatically formulate goals based on their knowledge and the given problem domain.

2. Problem Formulation:
   - After the goal is formulated, the agent needs to formulate the problem in a way that can be solved using search techniques.
   - This involves representing the problem as a search space, defining the initial state, possible actions, and a way to evaluate states (e.g., a heuristic function).
   - The problem formulation phase is crucial as it determines the efficiency and effectiveness of the subsequent search process.

Interface College of Computer Applications (ICCA)

3. Search:
   - Once the problem is formulated, the agent employs search algorithms to explore the search space and find a sequence of actions (solution) that leads to the goal state.
   - Various search algorithms can be used, such as breadth-first search, depth-first search, iterative deepening, greedy best-first search, or informed search algorithms like A* search.
   - The search process involves simulating sequences of actions and evaluating their outcomes until a solution is found or it is determined that no solution exists.
   - The search algorithm takes the formulated problem as input and outputs a sequence of actions (solution) if one exists.

4. Execution:
   - After a solution is found during the search phase, the agent can execute the recommended sequence of actions in the real environment or simulated environment.

- This execution phase involves carrying out each action in the solution sequence, one by one, while monitoring the results and updating the agent's state accordingly.
- If the execution is successful, the agent has solved the problem and achieved the goal.
- If the execution fails due to any unforeseen circumstances or changes in the environment, the agent may need to re-formulate the problem and search for a new solution.

We have a simple "formulate, search, execute" design for the agent

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  inputs: percept, a percept
  static: seq, an action sequence, initially empty
          state, some description of the current world state
          goal, a goal, initially null
          problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then do
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Note: - The SIMPLE-PROBLEM-SOLVING-AGENT function serves as a basic template or framework for designing problem-solving agents in artificial intelligence. Its purpose is to outline the general structure and behavior of such agents.

Well defined Problems and Solution

A formal definition of a problem consists of six components:

1. Initial State
2. Successor Function
3. Goal Test
4. Path Cost

1. Initial State

- It is the agent's starting state or initial step towards its goal. For example, if a taxi agent needs to travel to a location(B), but the taxi is already at location(A), the problem's initial state would be the location (A).

2. Successor Function

- This function takes a state as input and returns the set of (action, successor state) pairs that are possible from that state. It defines the legal actions and their resultant states.

State Space

- Together, the initial state and successor function implicitly define the state space of the problem-the set of all states reachable from the initial state. The state space is a conceptual representation or model that captures all possible states that the agent or

Interface College of Computer Applications (ICCA)

system could be in, for a given problem environment or domain. The state space forms a graph in which the nodes are states and the arcs between nodes are actions.

3. Goal Test

- A way to determine if a given state is a goal state that satisfies the problem's objective.
- There are two different ways in which the goal test component of a problem formulation can be specified:
  a. Explicit set of goal states:
     - In some problems, the goal states are explicitly enumerated as a set of specific states.
     - The goal test simply checks if the given state being evaluated is a member of this predefined goal state set.
     - For Example:- In the vacuum world, the goal is for the agent to clean all the locations, i.e., remove any dirt present in either location A or location B.
     - We can define the set of goal states explicitly as follows:
       Goal States = {(A, Clean, Clean), (B, Clean, Clean)}
     - This set contains two states:
     - (A, Clean, Clean) - The agent is in location A, and both locations A and B are clean.
     - (B, Clean, Clean) - The agent is in location B, and both locations A and B are clean.
     - In this formulation, the goal test function simply checks if the current state matches either of the two states in the Goal States set.
  b. Abstract property:
     - In other problems, the goal is not defined as a specific set of states. Instead, it is specified as an abstract property or condition that a state must satisfy to be considered a goal state.
     - For example, the game of chess, where the goal is not a particular board configuration, but rather the abstract condition of "checkmate." A state is a goal state if the opponent's king is under attack and cannot make any legal move to escape from the attack. This abstract property defines the goal, rather than an explicitly enumerated set of board configurations.

Interface College of Computer Applications (ICCA)

4. Path Cost:
   - A path in the state space is a sequence of states connected by a sequence of actions
- A function that assigns a numeric cost to each path (sequence of actions) taken through the state space. This allows evaluating paths and finding optimal solutions.
  Step Cost:
- The cost of taking a single action to transition between states. Often the path cost is the sum of step costs along the path.

Example Problems

- The problem solving approach has been used in a wide range of work contexts.
- There are two kinds of problem approaches
  - Standardized/ Toy Problem: Its purpose is to demonstrate or practice various problem solving techniques. It can be described concisely and precisely, making it appropriate as a benchmark for academics to compare the performance of algorithms.

- Real-world Problems: It is real-world problems that need solutions. It does not rely on descriptions, unlike a toy problem, yet we can have a basic description of the issue.

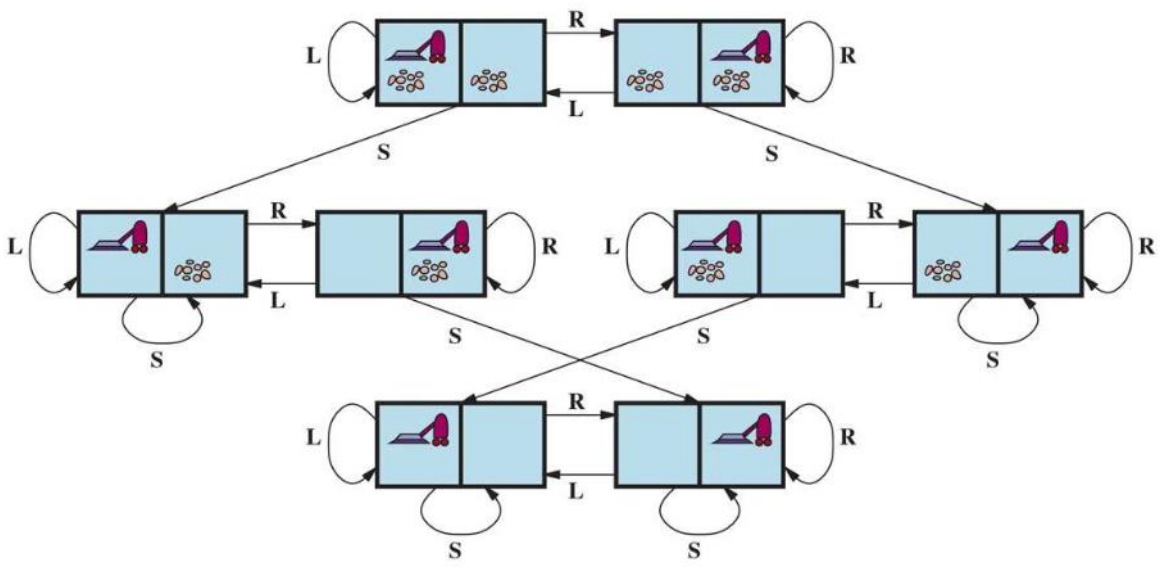
Some Standardized/Toy Problems

1. Vacuum World Problem
   Let us take a vacuum cleaner agent and it can move left or right and its job is to suck up the dirt from the floor.

The vacuum world's problem can be stated as follows:

- States: The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are 2 *2* 2= 8 possible world states.
- Initial state: Any state can be designated as initial state.
- Successor function: This generates the legal states that results from trying the three actions (left, right, suck). The complete state space is shown in above figure.
- Goal Test: This tests whether all the squares are clean.
- Path test: Each step costs on, so that the path cost is the number of steps in the path.

Vacuum World State Space



The state space graph for the two-cell vacuum world.

2. The 8 Puzzle Problem

- An 8-puzzle consists of a 3x3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The object is to reach the goal state, as shown in the below figure.

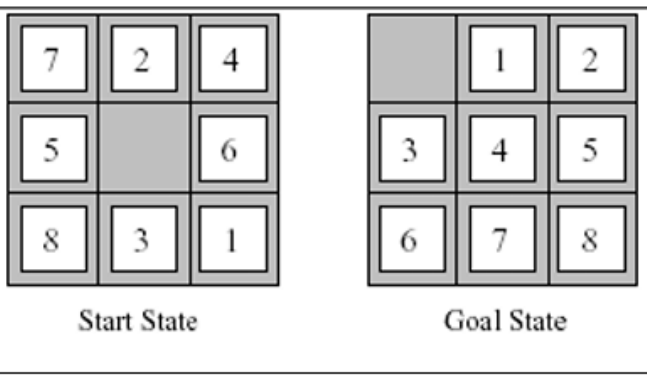


Figure 1.21 A typical instance of 8-puzzle.

The problem formulation is as follows :

- States: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- Initial state: Any state can be designated as the initial state. It can be noted that any given goal can be reached from exactly half of the possible initial states.
- Successor function: This generates the legal states that result from trying the four actions (blank moves Left,Right,Up or down).
- Goal Test: This checks whether the state matches the goal configuration shown in figure.
- Path cost: Each step costs 1, so the path cost is the number of steps in the path.

- The 8-puzzle belongs to the family of sliding-block puzzles, which are often used as test problems for new search algorithms110 in A.
- This general class is known as NP-complete.
- The 8-puzzle has 9!/2 = 181,440 reachable states and is easily solved.
- The 15 puzzle (4 x 4 board) has around 1.3 trillion states, an the random instances can be solved optimally in few milli seconds by the best search algorithms.
- The 24-puzzle (on a 5 x 5 board) has around 1025 states ,and random instances are still quite difficult to solve optimally with current machines and algorithms.

3. 8 – Queens Problem

Interface College of Computer Applications (ICCA)

- The 8-queens problem is a classic problem in computer science and artificial intelligence, where the goal is to place eight queens on an 8x8 chessboard in such a way that no queen attacks any other. In chess, a queen can attack any piece that is on the same row, column, or diagonal as her.
- The problem can be solved using various techniques, and one approach is to use state-space search algorithms.
- In this context, a state represents a particular arrangement of queens on the board, and the goal is to find a state where all eight queens are placed without any conflicts.

- We can formulate 8-Queens problem in two different ways :
  a. an incremental formulation
  b. a complete-state formulation.

Incremental Formulation:

- In the incremental formulation, the problem is approached by starting with an empty board and gradually adding queens one by one. The components of this formulation are:

- States: Any arrangement of 0 to 8 queens on the board is considered a state.
- Initial State: No queens are placed on the board initially.
- Successor Function: The successor function adds a single queen to any empty square on the board.
- Goal Test: The goal state is reached when 8 queens are placed on the board without any queen attacking another.

- The initial incremental formulation, has a very large state space of approximately 3 × 10^14 possible sequences to investigate.
- This is because every time a queen is placed, there are 64 possible choices for the next square, leading to a combinatorial explosion of possibilities.

Complete-state formulation:

- To reduce the state space and make the problem more manageable, an improved formulation is proposed.
- This formulation introduces constraints to avoid placing queens in squares that are already under attack by other queens.
- The components of this improved formulation are:
  - States: Arrangements of n queens (0 ≤ n ≤ 8), one per column in the leftmost columns, with no queen attacking another.
  - Successor Function: Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen already on the board.

- This improved formulation significantly reduces the state space for the 8-queens problem from approximately 3 × 10^14 states to just 2,057 states, making it much easier to find solutions.

Real World Example

1. Route – Finding Problem

- Route-finding problem is defined in terms of specified locations and transitions along links between them.
- Route-finding algorithms are used in a variety of applications, such as routing in computer networks, military operations planning, and airline travel planning systems.

Example:- Airline Travel Problem

- The airline travel problem is specifies as follows :
  - States: Each is represented by a location(e.g.,an airport) and the current time.
  - Initial State: The initial state is the airport you start from and the time you start your journey. This is provided as part of the problem specification.
  - Successor Function: The successor function generates the possible next states you can reach from your current state. It does this by considering all the scheduled flights that depart from your current airport after the current time plus the time it takes to transit within the airport (e.g., reaching the gate, going through security, etc.). Each of these flights will take you to a different airport at a different time, which becomes a new state.
  - Goal Test: The goal test checks if you have reached your desired destination airport by a specified time. If the current state satisfies this condition, it is considered the goal state.
  - Path cost: This depends upon the monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day ,type of air plane, frequent-flyer mileage awards, and so on.

2. Traveling Salesman Problem

Interface College of Computer Applications (ICCA)

- The Traveling Salesman Problem (TSP) is a classic combinatorial optimization problem in computer science and operations research. It can be defined as follows:
- Given a set of cities (or locations) and the distances (or costs) between each pair of cities, the TSP aims to find the shortest possible tour that visits each city exactly once and returns to the starting city.
- Let's consider the scenario where a travel blogger wants to visit all the major cities in the states of Rajasthan, Gujarat, and Maharashtra, starting and ending in Mumbai.
- The cities to be visited include:
    - Mumbai (Maharashtra)
    - Ahmedabad (Gujarat)
    - Surat (Gujarat)
    - Vadodara (Gujarat)
    - Jaipur (Rajasthan)
    - Jodhpur (Rajasthan)
    - Udaipur (Rajasthan)
    - Pune (Maharashtra)
    - Nashik (Maharashtra)
- The initial state would be "In Mumbai; visited {Mumbai}."
- A typical intermediate state could be "In Jaipur; visited {Mumbai, Ahmedabad, Surat, Vadodara, Jaipur}."
- The goal test would check whether the agent is back in Mumbai and all 9 cities have been visited.
- The actions correspond to travels between adjacent cities, where adjacency is determined by the road network connecting these cities. For example, from Mumbai, the travel blogger can directly travel to Ahmedabad, Pune, or Nashik.
- The state space consists of all possible combinations of the current location and the set of cities visited so far.
- The size of the state space grows exponentially with the number of cities, making it a challenging problem to solve optimally for larger instances.
- The travel blogger might want to find the shortest possible tour that satisfies the goal, taking into account the distances between cities, traffic conditions, and any specific preferences or constraints, such as visiting certain cities during daylight hours or avoiding certain roads.

3. VLSI layout

- A VLSI layout problem requires positioning millions of components and connections on a chip to minimize area ,minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield. The layout problem is split into two parts : cell layout and channel routing.

Interface College of Computer Applications (ICCA)

4. Robot navigation

- Robot navigation is a generalization of the route-finding problem. Rather than a discrete set of routes, a robot can move in a continuous space with an infinite set of possible actions and states.
- For a circular Robot moving on a flat surface, the space is essentially two dimensional.
- When the robot has arms and legs or wheels that also must be controlled, the search space becomes multi-dimensional. Advanced techniques are required to make the search space finite.

5. Automatic Assembly Sequencing

- The example includes assembly of intricate objects such as electric motors. The aim in assembly problems is to find the order in which to assemble the parts of some

objects. If the wrong order is chosen, there will be no way to add some part later without undoing some work already done.
- Another important assembly problem is protein design, in which the goal is to find a sequence of Amino acids that will be fold into a three-dimensional protein with the right properties to cure some disease.

6. Internet Searching
- In recent years there has been increased demand for software robots that perform Internet searching, looking for answers to questions, for related information, or for shopping deals. The searching techniques consider internet as a graph of nodes (pages) connected by links.

Searching for solutions

Search
- The process of looking for sequence of actions from the current state to reach the goal state is called search.
- The search algorithm takes a problem as input and returns a solution in the form of action sequence.

Search Tree
- Having formulated some problems, we now need to solve them. This is done by a search through the state space.
- A search tree is generated by the initial state and the successor function that together define the state space.
- Steps in searching process
  - Step 1:-Start with the initial state (the root node of the search tree)
  - Step 2:-Check if the current state is the goal state.
  - Step 3:-If the current state is the goal state, return the solution and stop.
  - Step 4:- Put the new states (successors) in the search queue or search tree for later consideration.
  - Step 5:-Choose one of the new states from the search queue or search tree using different search strategy(BFS,DFS,A*etc…) to become the new current state
  - Step 6:-Go back to Step 2 and repeat the process.

Interface College of Computer Applications (ICCA)

An informal description of the general tree-search algorithm.

```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  ioop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
```

- Each node in the search tree corresponds to a state in the state space.
- The arcs (edges) connecting the nodes in the search tree represent the transitions or actions that lead from one state to another state in the state space.

There are many ways to represent nodes, but we will assume that a node is a data structure with five components:

1. State: A state in the state space to which the node corresponds;
2. Parent-Node: The node in the search tree that generated this node;

3. Action: The action that was applied to the parent to generate the node;
4. Path-Cost: The cost, denoted by g(n), of the path from initial state to the node, as indicated by the parent pointers;
5. Depth: the number of steps along the path from the initial state.

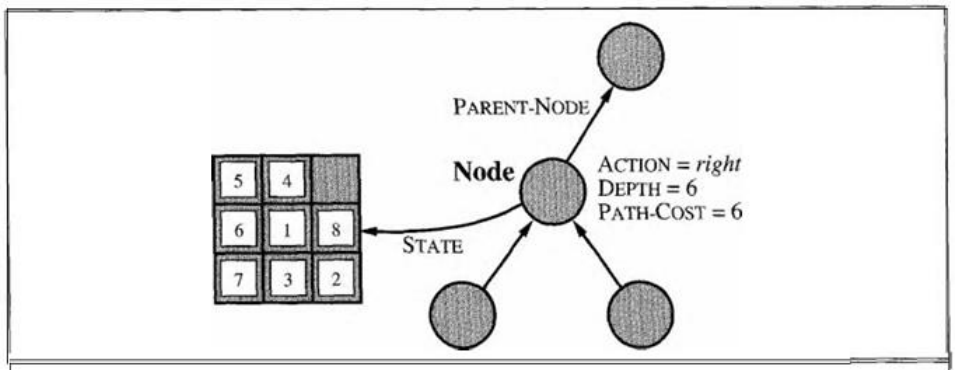


Figure 3.8 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

Note:- It is important to remember the distinction between nodes and states. A node is a book keeping data structure used to represent the search tree. A state corresponds to configuration of the world.

Fringe: - The fringe is a data structure (usually a queue or a stack) that holds the nodes (representing states) in a search tree that have been generated but not yet explored or expanded. These nodes are essentially the "frontier" or the "boundary" of the explored portion of the search tree.

The general Tree Search Algorithm

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure

  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
     if EMPTY?(fringe) then return failure
     node ← REMOVE-FIRST(fringe)
     if GOAL-TEST[problem] applied to STATE[node] succeeds
        then return SOLUTION(node)
     fringe ← INSERT-ALL(EXPAND(node, problem), fringe)

function EXPAND(node, problem) returns a set of nodes

  successors ← the empty set
  for each (action,result) in SUCCESSOR-FN[problem](STATE[node]) do
     s ← a new NODE
     STATE[s] ← result
     PARENT-NODE[s] ← node
     ACTION[s] ← action
     PATH-COST[s] ← PATH-COST[node] + STEP-COST(STATE[node], action, result)
     DEPTH[s] ← DEPTH[node] + 1
     add s to successors
  return successors
```

```
function Tree-Search(problem, fringe) returns a solution or failure
    // Initialize the fringe by creating a root node from the initial state
    root_node = Make-Node(Initial-State(problem))
    fringe = Insert(root_node, fringe)

    loop do
        // If the fringe is empty, there is no solution, so return failure
        if Empty?(fringe) then
            return failure

        // Remove the next node to be explored from the fringe
        node = Remove-First(fringe)

        // Test if the state of the current node is the goal state
        if Goal-Test(problem, State(node)) succeeds then
            // If it is the goal state, return the node as the solution
            return Solution(node)

        // If not the goal state, expand the node by generating its successors
        successors = Expand(node, problem)

        // Insert the successors into the fringe for later exploration
        fringe = Insert-All(successors, fringe)
    end loop

function Expand(node, problem) returns a set of nodes
    // Initialize an empty set to store the successor nodes
    successors = {}

    // For each possible action that can be applied to the current state
    for each (action, result) in Successor-Fn(problem, State(node)) do
        // Create a new node representing the resulting state
        s = a new NODE
        State(s) = result

        // Set the parent of the new node to the current node
        Parent-Node(s) = node

        // Set the action that led to the new node
        Action(s) = action

        // Calculate the path cost of the new node
        Path-Cost(s) = Path-Cost(node) + Step-Cost(State(node), action, result)

        // Set the depth of the new node
        Depth(s) = Depth(node) + 1

        // Add the new node to the set of successors
        add s to successors
    end for

    // Return the set of successor nodes
    return successors
```

Interface College of Computer Applications (ICCA)

Here's a detailed breakdown of the algorithm:

Tree-Search function

1. The function takes the problem and a fringe (a data structure like a queue or stack) as input.
2. It creates the root node from the initial state of the problem using the `Make-Node` function.
3. The root node is inserted into the fringe using the `Insert` function.
4. The algorithm enters a loop:
    a. First, it checks if the fringe is empty using the `Empty?` function. If the fringe is empty, it means no solution exists, so it returns `failure`.
    b. If the fringe is not empty, it removes the next node to be explored from the fringe using the `Remove-First` function. The order in which nodes are removed depends on the search strategy (e.g., FIFO for breadth-first, LIFO for depth-first).
    c. It checks if the state of the current node is the goal state using the `Goal-Test` function, which takes the problem and the state of the node as input. If it is the goal state, it returns the node as the solution using the `Solution` function.
    d. If the current node is not the goal state, it expands the node by calling the `Expand` function, passing the node and the problem as input. The `Expand` function returns a set of successor nodes.
    e. The set of successor nodes is inserted into the fringe using the `Insert-All` function, so they can be explored later.
5. The loop continues until a solution is found (returned in step 4c) or the fringe becomes empty (failure returned in step 4a).

Expand function

1. The function takes a node and the problem as input.
2. It initializes an empty set called `successors` to store the successor nodes.
3. For each possible action that can be applied to the state represented by the input node:
    a. It creates a new node `s` using the `a new NODE` statement.
    b. It sets the state of `s` to the result of applying the action to the parent node's state using the `Successor-Fn` function, which returns a set of (action, result) pairs.
    c. It sets the parent of `s` to the input node using the `Parent-Node` function.
    d. It sets the action that led to `s` using the `Action` function.
    e. It calculates the path cost of `s` as the path cost of the parent node plus the step cost of the action, using the `Path-Cost` and `Step-Cost` functions.
    f. It sets the depth of `s` as the depth of the parent node plus 1 using the `Depth` function.
    g. It addds `s` to the `successors` set.
4. After iterating through all possible actions, the function returns the set of `successors`.

Interface College of Computer Applications (ICCA)

<u>Measuring Problem – Solving Performance (Properties of search strategy)</u>

- The output of problem-solving algorithm is either failure or a solution. (Some algorithms might struck in an infinite loop and never return an output.
- The algorithm's performance can be measured in four ways :
    - Completeness : Is the algorithm guaranteed to find a solution when there is one?
    - Optimality : Does the strategy find the optimal solution
    - Time complexity : How long does it take to find a solution?
    - Space complexity : How much memory is needed to perform the search?

Search Strategy

- Two broad categories of search strategies used in problem-solving and artificial intelligence.
    1. Uninformed Search
    2. Informed Search (Heuristic Search)

Uninformed Search

- Uninformed search strategies, also known as blind search strategies, they do not use any additional information or heuristics to guide the search process.
- They explore the search space systematically, without any knowledge about the problem domain or the proximity of the nodes to the goal state.
- These strategies rely solely on the problem's definition and the current state to determine the next steps.
- Examples of uninformed search strategy are
    - Breadth-First Search(BFS)
    - Uniform-Cost Search
    - Depth-First Search
    - Depth-limited Search
    - Iterative deepening depth first search
    - Bidirectional Search

1. Breadth-First Search(BFS)

- Breadth-first search is a strategy for exploring a tree or graph data structure. It starts at the root node (or initial state) and explores all the neighbouring nodes first before moving to the next level of nodes.

Working of BFS

1. Start with the root node and add it to a queue.
2. Take the node from the front of the queue and explore its neighbouring nodes.
3. Add all the unexplored neighbouring nodes to the end of the queue.
4. Repeat steps 2 and 3 until the queue is empty or the goal is found.

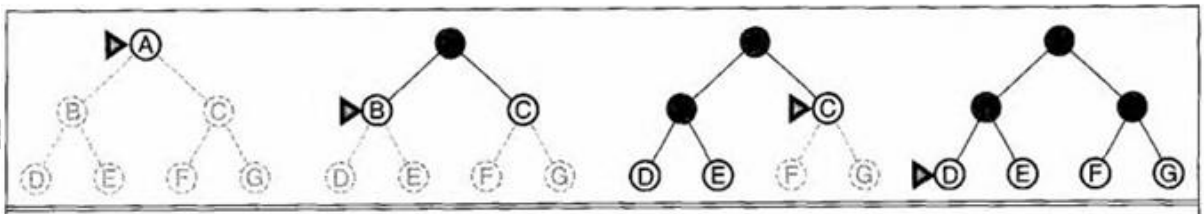


Figure 3.10 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

- The key idea is that the nodes are explored level by level, starting from the root.
- First, all the nodes at the current level are explored, and then the algorithm moves to the next level.
- This process continues until the goal is found or all nodes have been explored.
- The queue data structure ensures that the nodes are explored in the order they were added to the queue, which is the breadth-first order.
- The first node added to the queue will be the first one explored, and the last node added will be the last one explored.
- This approach is particularly useful when the solution is relatively shallow (closer to the root) because it explores all the nodes at a given depth before moving deeper into the tree or graph.

Algorithm for BFS

```
Function BFS(start_node, goal_node):
    Initialize an empty queue
    Initialize an empty set for visited nodes
    Enqueue the start_node in the queue
    Add the start_node to the visited set

    while the queue is not empty:
        current_node = Dequeue from the queue

        if current_node is the goal_node:
            return True (goal found)

        for each neighbor_node of current_node:
            if neighbor_node is not in the visited set:
                Enqueue neighbor_node in the queue
                Add neighbor_node to the visited set

    return False (goal not found)
```

Properties of BFS

1. Completeness: - Yes (if b is finite): The breadth-first search algorithm is complete if the branching factor (b) of the search tree is finite..

2. Time Complexity: - b + b^2 + b^3 + ... + b^d + b(b^d+1 - b) = O(b^(d+1)), i.e., exp. in d: The time complexity of breadth-first search is exponential in the depth of the solution (d). The formula represents the number of nodes expanded at each level, summed up to the depth of the solution. The time complexity is O(b^(d+1)), which means it grows exponentially with the depth of the solution.

3. Space Complexity: - O(b^(d+1)) (keeps every node in memory): The space complexity of breadth-first search is also exponential in the depth of the solution, O(b^(d+1)). This is because the algorithm keeps all the generated nodes in memory, including the ones that have not been expanded yet.

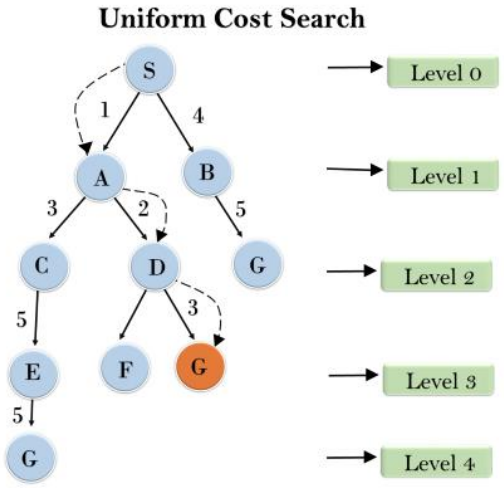
Interface College of Computer Applications (ICCA)

4. Optimality: -No, unless step costs are constant: Breadth-first search is not guaranteed to find the optimal solution unless the step costs (the costs of moving from one state to another) are constant. If step costs vary, the algorithm may find a suboptimal solution.

Note:- Space is the big problem; can easily generate nodes at 100MB/sec, so 24hrs = 8640GB: The main limitation of breadth-first search is its memory requirement.

2. Uniform Cost Search

- Breadth-first search is optimal when all step costs are equal, because it always expands the shallowest unexpanded node.
- UCS is a best-first search algorithm that expands the node with the lowest path cost g(n) first, where g(n) is the cost of the path from the initial state to node n.
- UCS guarantees to find the optimal solution to the problem provided that the cost function is non-negative.
- A uniform-cost search algorithm is implemented by the priority queue.
- It gives maximum priority to the lowest cumulative cost. Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.



Algorithm for UCS

```
Function UCS(start_node, goal_node):
    Initialize an empty priority queue (PQ)
    Initialize a set for visited nodes (visited)
    Initialize a dictionary to store the cost of reaching each node (cost)

    Enqueue start_node in PQ with cost 0
    cost[start_node] = 0

    while PQ is not empty:
        current_node = Dequeue the node with the minimum cost from PQ

        if current_node is the goal_node:
            return cost[current_node]  # Return the cost of reaching the goal

        if current_node not in visited:
            visited.add(current_node)

            for neighbor_node, edge_cost in neighbors(current_node):
                new_cost = cost[current_node] + edge_cost
                if neighbor_node not in cost or new_cost < cost[neighbor_node]:
                    cost[neighbor_node] = new_cost
                    Enqueue neighbor_node in PQ with cost new_cost

    return false
```

Interface College of Computer Applications (ICCA)

Properties of Uniform Cost Search

1. Completeness: If there is a solution, uniform cost search will find it, provided that the step costs are non-negative.
2. Optimality: If there is a solution, uniform cost search will find the least-cost solution path, provided that the step costs are non-negative.
3. Time and Space Complexity: The time and space complexity of uniform cost search is O(b^(1+⌊C*/ε⌋)) where b is the branching factor, C* is the cost of the optimal solution, and ε is the minimum non-zero step cost. This complexity is exponential in the solution cost, making uniform cost search impractical for problems with very large solution costs.

3. Depth First Search

- DFS is an uninformed search strategy that explores the deepest node in the current frontier (fringe) of the search tree.
- It proceeds immediately to the deepest level of the search tree, expanding nodes along a single path as far as possible.
- When a node with no successors is encountered (a leaf node), the search backtracks to the nearest ancestor with unexplored successors.
- This strategy is implemented using a Last-In-First-Out (LIFO) queue, commonly known as a stack data structure.

Working of DFS

1. Start at the root node.
2. Push the root node onto a stack.
3. While the stack is not empty:
    a. Pop the top node from the stack and mark it as visited.
    b. For each unvisited neighbor of the popped node, push the neighbor onto the stack.
4. Once all nodes have been visited or the target node is found, the search is complete.

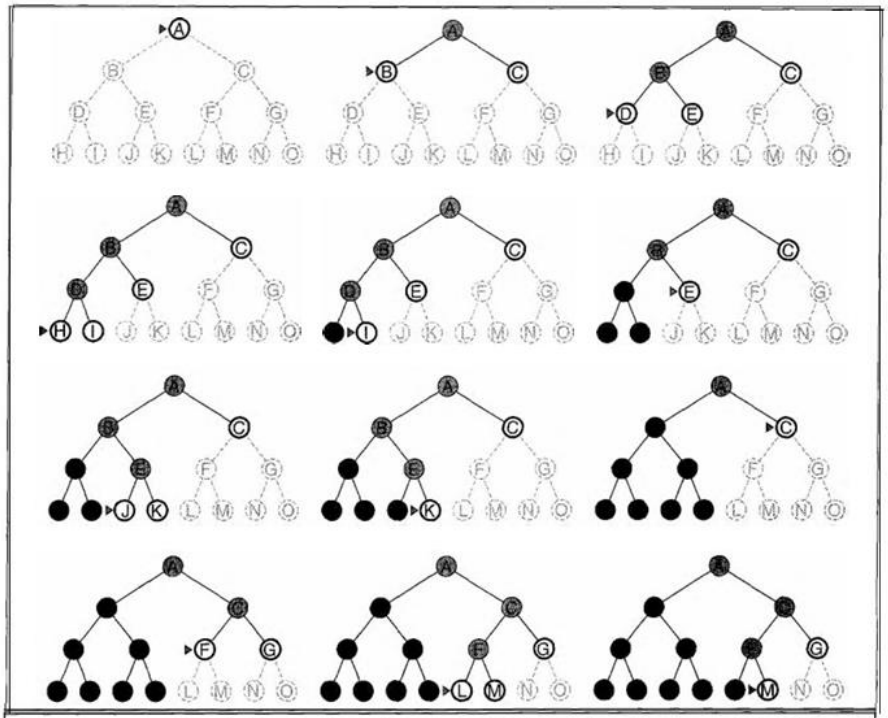


Figure 3.12 Depth-first search on a binary tree. Nodes that have been expanded and have no descendants in the fringe can be removed from memory; these are shown in black. Nodes at depth 3 are assumed to have no successors and M is the only goal node.

I ons (ICCA)

Algorithm for DFS

```
Function DFS(start_node, goal_node):
    Initialize an empty set visited

    Stack = [start_node]
    visited.add(start_node)

    while Stack is not empty:
        current_node = Stack.pop()
```

```
        if current_node is goal_node:
            return True  # Goal found

        for neighbor_node in neighbors(current_node):
            if neighbor_node not in visited:
                visited.add(neighbor_node)
                Stack.append(neighbor_node)

    return False  # Goal not found
```

Properties of DFS

1. Completeness: DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.
2. Time Complexity: Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by: T(n)= 1+ n2+ n3 +.....+ nm=O(n^m)
   Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)
3. Space Complexity: DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is O(bm).
4. Optimal: DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

Drawback of Depth-first-search

- The drawback of depth-first-search is that it can make a wrong choice and get stuck going down very long(or even infinite) path when a different choice would lead to solution near the root of the search tree. For example ,depth-first-search will explore the entire left subtree even if node C is a goal node.

Note:- Backtracking Search is a variant of depth-first search called backtracking search uses less memory and only one successor is generated at a time rather than all successors.; Only O(m) memory is needed rather than O(bm)

4. Depth Limited Search

- Depth-limit search, also known as depth-bounded search or depth-limited search, is a graph search algorithm that restricts the depth or level of nodes to be explored in the search tree.
- It is a modification of the general depth-first search (DFS) algorithm, where a maximum depth limit is set to prevent the search from going indefinitely deep into the tree.

Working of DLS

1. Start with the root node of the search tree.
2. Define a maximum depth limit (depth_limit).
3. Explore the nodes by expanding the current node's successors in a depth-first manner until the depth limit is reached or the goal node is found.
4. If the depth limit is reached, backtrack to the previous level and continue exploring other branches.
5. If the goal node is found before reaching the depth limit, return the solution path.
6. If all nodes have been explored without finding the goal, report that no solution was found.

Recursive Implementation of DLS

```
function Depth-Limited-Search( problem, limit) returns a solution/fail/cutoff
return Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit)
function Recursive-DLS(node, problem, limit) returns solution/fail/cutoff
cutoff-occurred?← false
if Goal-Test(problem,State[node]) then return Solution(node)
else if Depth[node] = limit then return cutoff
else for each successor in Expand(node, problem) do
result← Recursive-DLS(successor, problem, limit)
if result = cutoff then cutoff_occurred?← true
else if result not = failure then return result
if cutoff_occurred? then return cutoff else return failure
```

```
function Depth-Limited-Search(problem, limit):
    return Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit)

function Recursive-DLS(node, problem, limit):
    cutoff-occurred? ← false
    if Goal-Test(problem, State[node]) then
        return Solution(node)
    else if Depth[node] = limit then
        return cutoff
    else
        for each successor in Expand(node, problem) do
            result ← Recursive-DLS(successor, problem, limit)
            if result = cutoff then
                cutoff-occurred? ← true
            else if result ≠ failure then
                return result
        if cutoff-occurred? then
            return cutoff
        else
            return failure
```

Properties of Depth Limit Seach

1. Completeness: DLS search algorithm is complete if the solution is above the depth-limit.
2. Time Complexity: Time complexity of DLS algorithm is O(b^ℓ).
3. Space Complexity: Space complexity of DLS algorithm is O(b×ℓ).
4. Optimal: Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if ℓ>d.

Interface College of Computer Applications (ICCA)

Iterative Deepening Depth First Search

- IDDFS is a combination of both Depth first and Breadth first Search.
- IDDFS is a strategy that applies depth-first search repeatedly, increasing the depth limit with each iteration until the goal is found or the entire search space is exhausted.
- It effectively performs a breadth-first traversal of the search tree by incrementing the depth limit, while maintaining the memory efficiency of depth-first search by exploring one branch at a time.

Working of IDDFS

- It starts with a depth limit of 0, which means only the root node is visited.

- If the goal node is not found at the current depth limit, the depth limit is increased by 1, and a new depth-limited search is performed.
- This process continues, gradually increasing the depth limit until either the goal node is found or the entire search space has been explored (when the depth limit becomes larger than the maximum depth of the search tree).

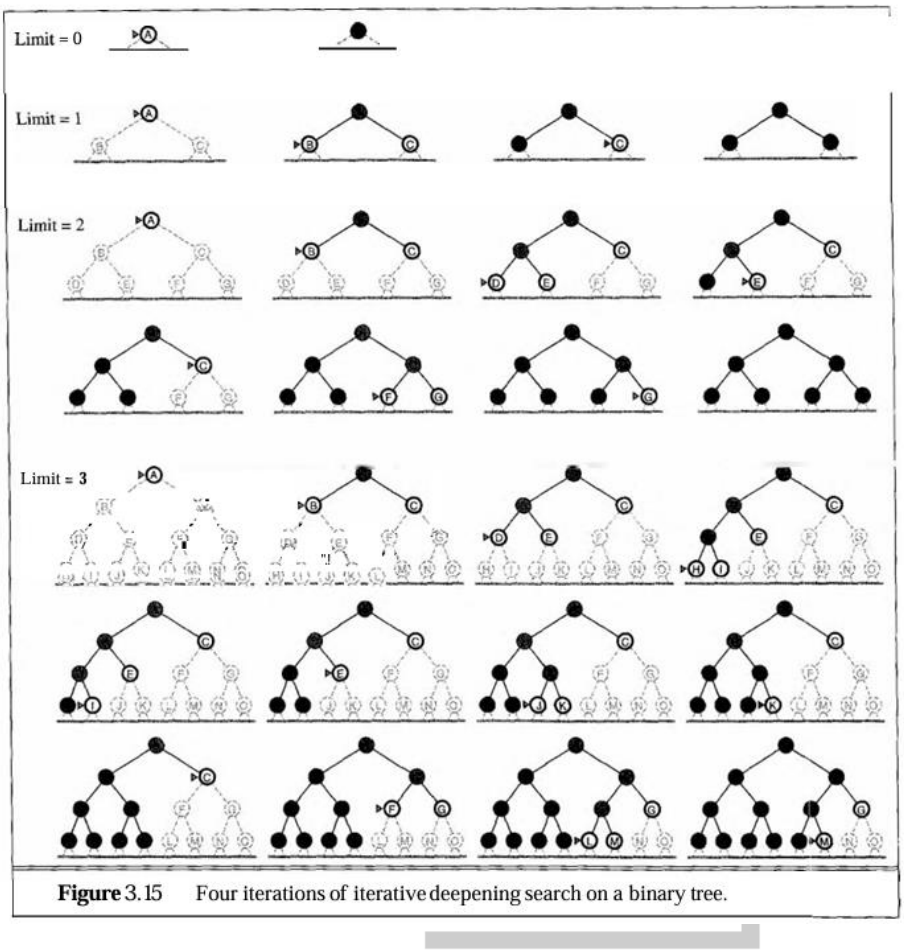


Figure 3.15 Four iterations of iterative deepening search on a binary tree.

Algorithm for IDDFS

```
Function IDDFS(start_node, goal_node):
    for depth_limit from 0 to infinity:
        result = DLS(start_node, goal_node, depth_limit)
        if result is not None:
            return result
    return None  # Goal not found

DLS(node, goal_node, depth_limit):
    if depth_limit == 0 and node != goal_node:
        return None  # Base case: depth limit reached without finding the goal
    if node == goal_node:
        return node  # Goal found

    for neighbor_node in neighbors(node):
        result = DLS(neighbor_node, goal_node, depth_limit - 1)
        if result is not None:
            return result
    return None  # Goal not found
```

Interface College of Computer Applications (ICCA)

Properties of IDDFS

1. Completeness: IDDFS is a complete algorithm, meaning that if a solution exists, it will find it by gradually increasing the depth limit and eventually exploring the entire search space.
2. Time Complexity: The time complexity of IDDFS is O(b^d), where b is the branching factor (maximum number of successors for any node), and d is the depth of the shallowest goal node. This exponential time complexity is due to the fact that IDS explores all nodes up to depth d in the worst case.
3. Space Complexity: The space complexity of IDDFS is O(bd), which is the maximum depth of the recursion stack, corresponding to the depth limit at which the goal node is found (or the maximum depth if no goal is found).
4. Optimality: IDS is not optimal for finding the shortest path unless the step costs are constant. If the costs of transitions between nodes vary, IDDFS may not find the optimal (least-cost) path to the goal node.

Bi-directional Search

- Bidirectional search is a graph search algorithm that searches for a solution path from both the initial state and the goal state, simultaneously.
- It is an optimization over other uninformed search algorithms like breadth-first search (BFS) and depth-first search (DFS).
- The key idea behind bidirectional search is to reduce the search space by exploring from both ends, potentially finding a solution faster than searching from one end alone.

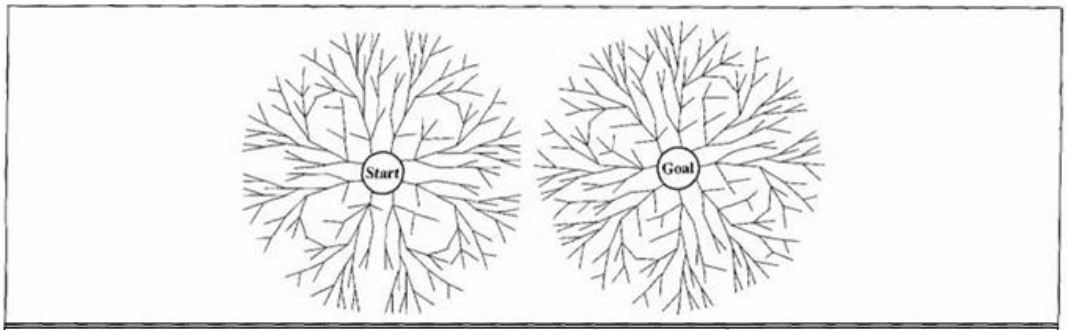


Figure 3.16 A schematic view of a bidirectional search that is about to succeed, when a branch from the start node meets a branch from the goal node.

Interface College of Computer Applications (ICCA)

Properties of Bidirectional Search

1. Completeness: Bidirectional search is complete if the underlying search algorithm used for each direction (e.g., BFS or DFS) is complete. If a solution exists, bidirectional search will find it.
2. Time Complexity:- b is the branching factor(the maximum number of successors of any node) of the tree, and distance between the start and end vertex is d, normal BFS/DFS complexity is O(b^d). In the case of Bidirectional Search, we run two simultaneous search operations with the complexity of each operation as O(b^(d/2)) which makes the total complexity as O(b^(d/2)+b^(d/2)). This clearly is significantly less than O(b^d).
3. Space Complexity: The space complexity of bidirectional search is the sum of the space complexities of the two searches being performed in each direction. For example, if BFS is used in both directions, the space complexity is O(b^(d/2)), where b is the branching factor, and d is the depth of the solution.
4. Optimality: It is optimal if BFS is used for search and paths have uniform cost.

Informed Search (heuristic Search)

- Informed search, also known as heuristic search, is a type of search algorithm that uses additional information, known as a heuristic function(heuristic value), to guide the search towards more promising areas of the search space.
- Heuristic functions are used to estimate the distance or cost from a given state to the goal state.
- The heuristic function is used to guide the search by selecting the next state to explore based on the estimated distance to the goal.

Important concept in heuristic Search

1. Heuristic Function (h(n)):
   - A heuristic function estimates the cost from the current node to the goal node. It provides additional information about the problem to guide the search.
2. Evaluation Function (f(n)):
   - The evaluation function combines the actual cost to reach the current node and the estimated cost to reach the goal. For example, in the A* algorithm, the evaluation function is:

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

     Where, g(n) = Actual cost

     h(n) = Heuristic Value

Heuristics Function

- A heuristic function is a key component of Informed Search algorithms, such as Best-First Search, Greedy Search, and A* Search.
- It is a function that estimates the cost or distance from a given state or node to the goal state.

Properties of Heuristic Function

- Admissibility: A heuristic function is admissible if it never overestimates the actual cost or distance to the goal.
- Consistency (or Monotonicity): A heuristic is consistent if, for every node n and every successor n′ of n, the estimated cost of reaching the goal from n is no greater than the cost of getting to n′ plus the estimated cost from n′ to the goal.
- Accuracy: A heuristic function should provide as accurate an estimate as possible for the remaining cost to the goal. The more accurate the heuristic function, the more efficiently the search algorithm can find the optimal solution.
- Computational Efficiency: The heuristic function should be computationally inexpensive to evaluate, as it will be called repeatedly during the search process.

Interface College of Computer Application (ICCA)

Some examples of Heuristic Functions

1. Euclidean Distance(Straight line Distance):
   - Used in pathfinding problems where the distance is the straight-line distance between two points. It's commonly used in grid-based maps.
   - Heuristic function: h(n) = sqrt((x_n - x_goal)^2 + (y_n - y_goal)^2)
   - Where (x_n, y_n) is the position of node n, and (x_goal, y_goal) is the position of the goal node.

2. Manhattan Distance:
   - In the sliding tile puzzle, where the goal is to arrange tiles in a specific order by sliding them horizontally or vertically, a common heuristic is the sum of the Manhattan distances between each tile's current position and its desired position.
   - Heuristic function: h(n) = Σ |x_i - x_goal_i| + |y_i - y_goal_i|

- Where (x_i, y_i) is the current position of tile i, and (x_goal_i, y_goal_i) is the desired position of tile i.

3. Misplaced Tiles:

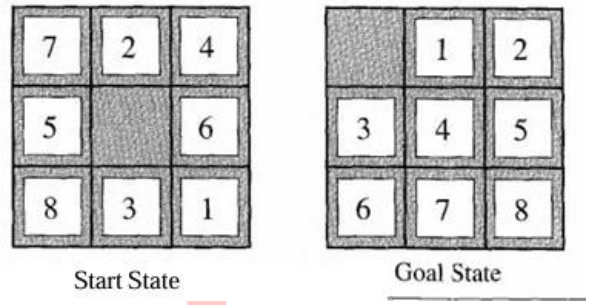
- The misplaced tiles heuristic is a popular heuristic function used in the sliding tile puzzle problem.
- The misplaced tiles heuristic counts the number of tiles that are not in their correct positions in the goal state.
- Heuristic function: h(n)=number of misplaced tiles.

Example

8 Puzzle problem

The two common heuristic functions used to solve 8 puzzle problem is

1. Misplaced tiles



h(n) = Number of misplaced tiles

- For above start state, all of the 8 tiles are out of position, so the heuristic function h(n) = 8.

2. Manhattan distance(the city block distance)

h(n) = The sum of the distances of the tiles from their goal positions.

- For the above start state, the Manhattan distance is
h(n) = 3+1+2+2+2+3+3+2 = 18

Best First Search

- The general approach we will consider is called best-first search. Best-first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an evaluation function, f (n).
- Traditionally, the node with the lowest evaluation is selected for expansion, because the evaluation measures distance to the goal.
- Best-first search can be implemented within our general search framework via a priority queue, a data structure that will maintain the fringe in ascending order of f -values.
- There are two well-known variants of Best-First Search:
  - Greedy Best-First Search (GBFS)
  - A* Search.

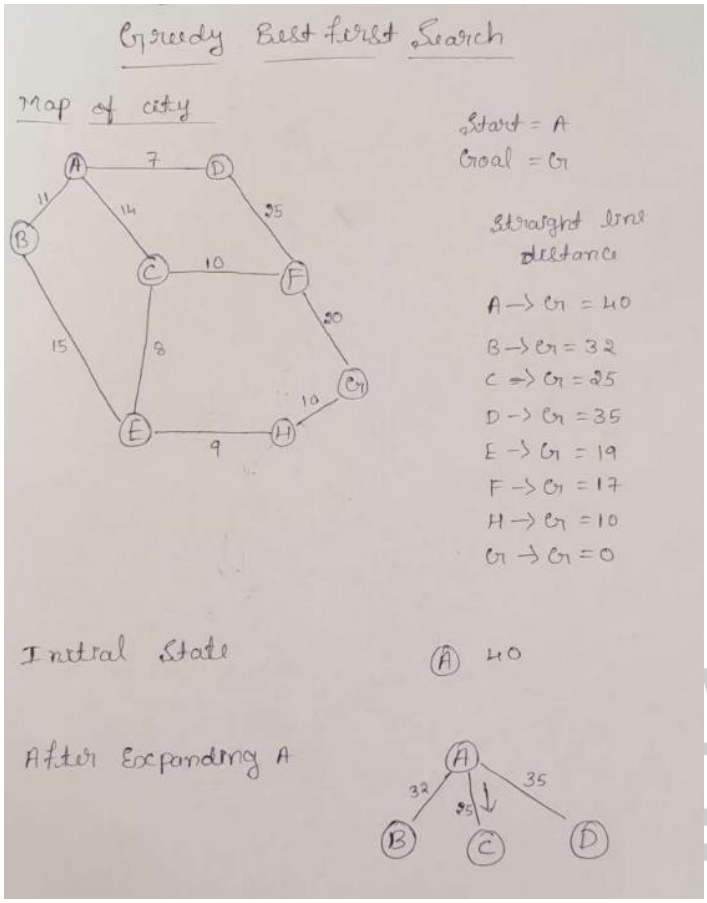
Greedy Best- First Search

- Greedy Best-First Search is a specific type of Best-First Search where the evaluation function is purely heuristic-based.
- GBFS uses a heuristic function h(n) to estimate the cost from the current node n to the goal, and it always expands the node with the lowest heuristic value first.

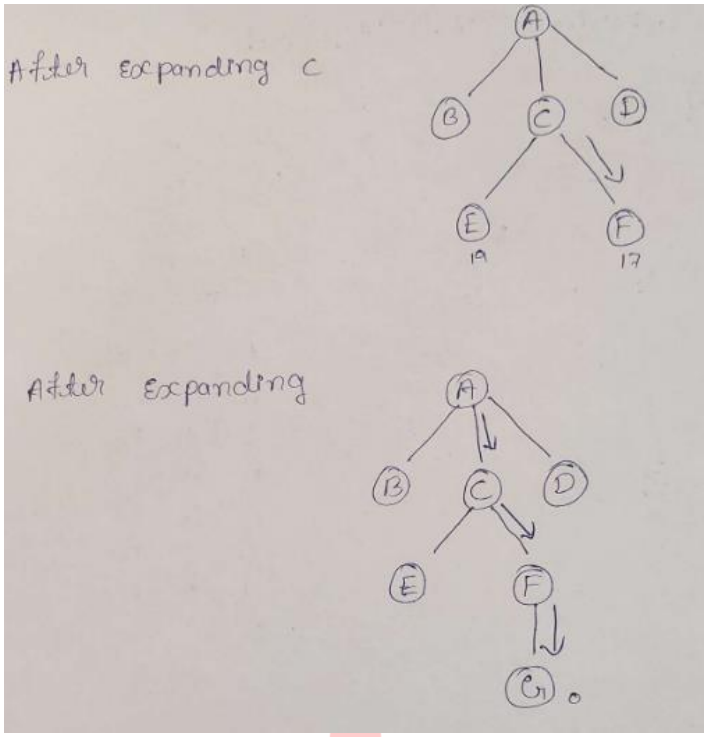
Interface College of Computer Applications (ICCA)

Working of GBFS

1. Initialize the priority queue with the start node and its heuristic value.
2. While the priority queue is not empty: a. Remove the node with the lowest heuristic value from the priority queue. b. If the removed node is the goal node, return the solution path. c. Otherwise, generate the successors of the removed node. d. For each successor, calculate its heuristic value and insert it into the priority queue.
3. If the priority queue becomes empty and the goal node has not been found, return failure.



Interface College of Computer Applications (ICCA)



Algorithm of GBFS

```
Function GreedyBestFirstSearch(start_node, goal_node):
    open_list = {start_node} #priority_Queue
    closed_set = {}

    while open_list is not empty:
        current = node from open_list with the lowest heuristic value
        remove current from open_list
        add current to closed_set

        if current == goal_node:
            return current  # Goal found

        for neighbor_node in neighbors(current):
            if neighbor_node not in closed_set and neighbor_node not in open_list:
                add neighbor_node to open_list

    return None  # Goal not found
```

Interface College of Computer Applications (ICCA)

Properties of GBFS

1. Time Complexity:
   - In the worst case, GBFS may explore all nodes in the search space, leading to a time complexity of O(b^m), where b is the branching factor (maximum number of successors for a node), and m is the maximum depth of the search tree.
   - However, if the heuristic function is well-designed and guides the search effectively, GBFS can be more efficient than other uninformed search algorithms like Breadth-First Search or Depth-First Search.
2. Space Complexity:
   - The space complexity of GBFS is O(b^m), as in the worst case, it may need to store all nodes in the search tree.

- However, in practice, the space complexity depends on the number of nodes stored in the priority queue and the memory required to represent each node.

3. Completeness:
   - GBFS is not complete, meaning it is not guaranteed to find a solution if one exists, even with an admissible heuristic function.
   - GBFS can get stuck in infinite loops or fail to find a solution if there are "loops" or "plateaus" in the search space where the heuristic value does not change, causing the algorithm to explore the same nodes repeatedly.
4. Optimality:
   - GBFS is not optimal, meaning it does not guarantee to find the optimal solution (the one with the lowest cost) even if one exists.
   - Since GBFS only considers the heuristic estimate and ignores the actual cost of reaching the current node from the start, it can get trapped in suboptimal paths.

A* Search

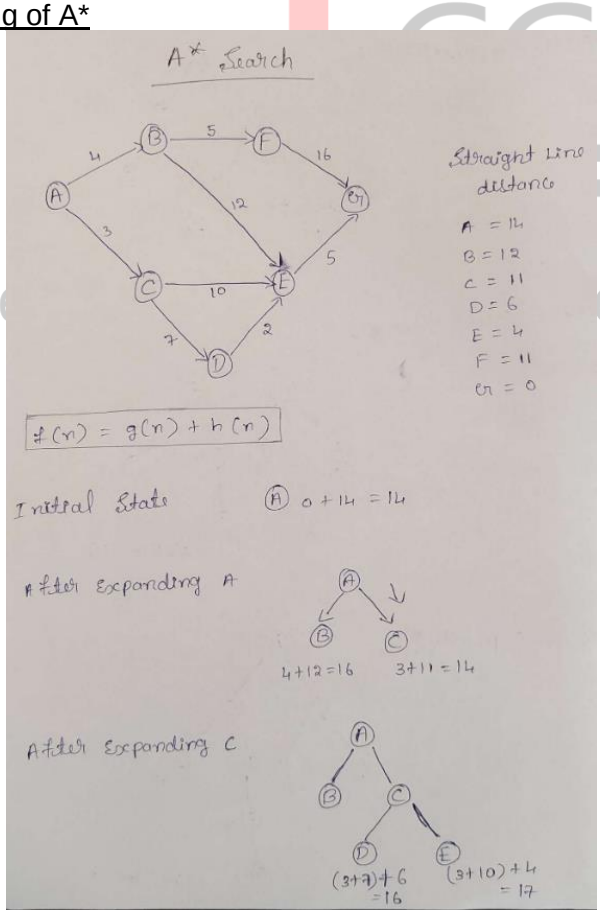
- A* is a best-first search algorithm that uses heuristics to guide its search toward the most promising paths.
- A* combines the benefits of Dijkstra's algorithm and Greedy Best-First-Search to efficiently find the shortest path from a start node to a target node.
- A* uses the following evaluation function to find the minimum cost

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

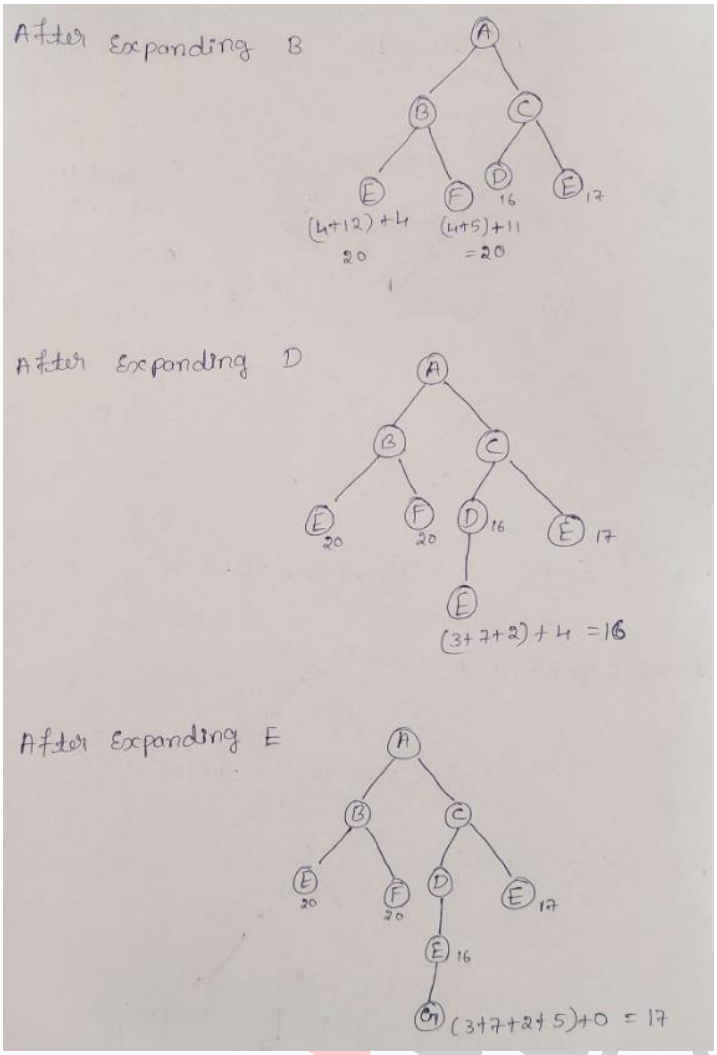
where,

f(n): Total estimated cost from start to goal through node n
g(n): Actual cost from start to node n
h(n): Heuristic estimate from node n to goal

Working of A*



Int er Applications (ICCA)



Algorithm for A*

```
Function AStar(start_node, goal_node):
    open_list = {start_node}
    closed_set = {}
    g_score = {start_node: 0}  # Cost from start to node
    f_score = {start_node: Heuristic(start_node, goal_node)}  # Estimated total cost

    came_from = {}  # To reconstruct path

    while open_list is not empty:
        current = node from open_list with the lowest f_score

        if current == goal_node:
            return ReconstructPath(came_from, current)

        remove current from open_list
        add current to closed_set

        for neighbor_node in Neighbors(current):
            if neighbor_node in closed_set:
                continue  # Skip nodes already evaluated
```

Interface College of Computer Applications (ICCA)

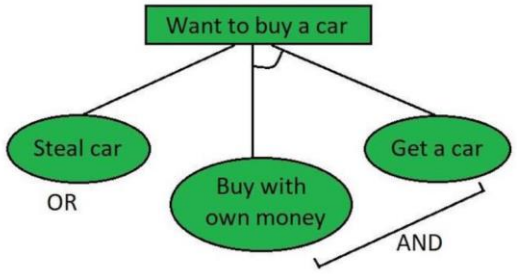
```
            tentative_g_score = g_score[current] + Cost(current, neighbor_node)

            if neighbor_node not in open_list:
                add neighbor_node to open_list
            elif tentative_g_score >= g_score[neighbor_node]:
                continue  # This path is not better

            # This path is the best so far, record it
            came_from[neighbor_node] = current
            g_score[neighbor_node] = tentative_g_score
            f_score[neighbor_node] = g_score[neighbor_node] +
                                     Heuristic(neighbor_node, goal_node)

        return None  # No path found
```

Properties of A*

1. Time Complexity:
    a. Worst Case: O(b^d)
    b. Best Case: O(d)
    c. Average Case: Depends on the heuristic function
2. Space Complexity: O(b^d) in the worst case
3. Optimality: Guaranteed if the heuristic is admissible and consistent
4. Completeness: Guaranteed if the search space is finite or there is a finite path to the goal and the heuristic is admissible.

AO* Search

- The AO* algorithm is based on AND-OR graphs to break complex problems into smaller ones and then solve them.
- The AND side of the graph represents those tasks that need to be done with each other to reach the goal, while the OR side stands alone for a single task.



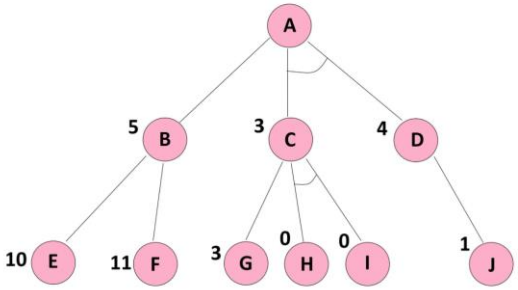
AND-OR Graph

- In the above example, we can see both tasks of the AND side, which are connected by an AND-ARCS, need to be done to have a car, while the OR side lets having a car just by a single task of stealing.
- In general, in a graph for each node, there could be multiple OR and AND sides, where each AND side itself may have multiple successor nodes connected to each other by an AND-ARCS.

Working of AO*

- AO* works based on this formula: f(N)=g(N)+h(N),
    - Where, g(N) is the actual cost of going from the starting node to the current node,
    - h(N) is the estimated or heuristic cost of going from the current node to the goal node,
    - f(N) is the actual cost of going from the starting node to the goal node.

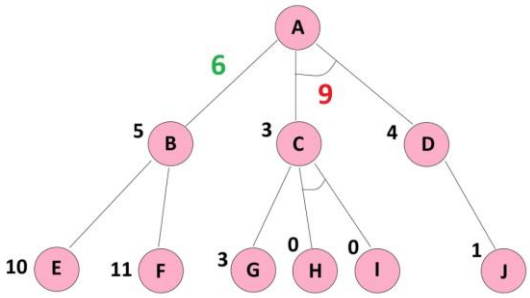
Let's see an example below to explain AO* step by step to find the lowest cost path from the starting node A to the goal node:



- It should be noted that the cost of each edge is the same as 1, and the heuristic cost to reach the goal node from each node of the graph is shown beside it.

Step 1

- First, we begin from node A and calculate each of the OR side and AND side paths. The OR side path f(A-B)=g(B)+h(B) = 1+5=6, where 1 is the cost of the edge between A and B, and 5 is the estimated cost from B to the goal node.
- The AND side path f(A-CD) = g(c)+h(c)+ g(d)+h(d) = 1+3+1+4=9, where the first 1 is the cost of the edge between A and C, 3 is the estimated cost from C to the goal node, the second 1 is the cost of the edge between A and D, and 4 is the estimated cost from D to the goal node. Let's see an update in below figure.

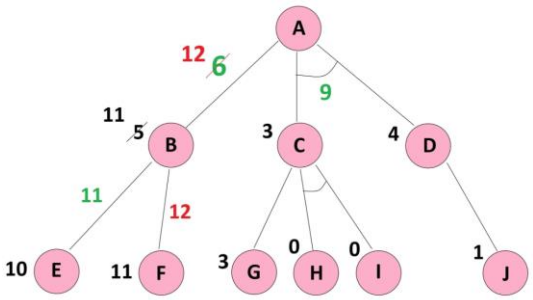


- Since the cost of f(A-B) is the minimum cost path, we proceed on this path in the next step.

Step 2

Interface College of Computer Applications (ICCA)

- In this step we continue on the f(A-B) from B to its successor nodes i.e., E and F, where f(B-E) = 1+10 = 11 and f(B-F)=1+11 = 12. Here, f(B-E) has a lower cost and would be chosen.
- Now, we have reached the bottom of the graph where no more level is given to add to our information.
- Therefore, we can do the backpropagation and correct the heuristics of upper levels. In this vein, the updated f(B) = f(B-E) = 11, and as a consequence the updated f(A-B) = g(B)+updated_f(B) = 1+11 = 12 . Let's see an update in below figure.

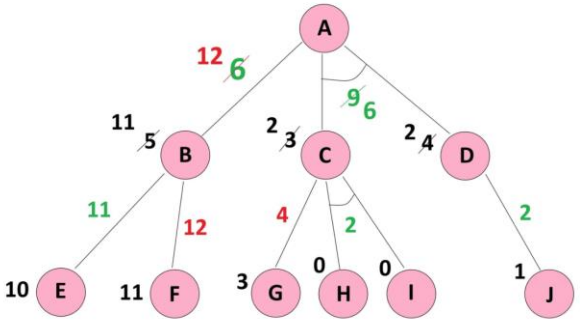


- Now, we can see that f(A-CD) with a cost of 9 is lower than the updated f(A-B) with a cost of 12. Therefore, we need to proceed on this path to find the minimum cost path from A to the goal node.

Note:- If the updated f(A-B) had a lower cost than f(A-CD) , then we were done, and no more calculations would be required.

Step 3

- In this step, we do the calculations for the AND side path, i.e., f(A-CD) , and first explore the paths attached to node C. In this node again we have an OR side where f(C-G)=1+3=4, and an AND side where f(C-HI) = 1+0+1+0 = 2, and as a consequence the updated f(C) =2.
- Also, the updated F(D) = 2, since f(D-J) = 1+1 = 2. By these updated values for f(C) and F(D), the updated f(A-CD) =1+2+1+2=6. Let's see an example below:



- This updated f(A-CD) with the cost of 6 is still less than the updated f(A-B) with the cost of 12, and therefore, the minimum cost path from A to the goal node goes from f(A-CD) by the cost of 6.

Properties of AO*

1. Time Complexity:
   - The time complexity of an algorithm describes the computational time it takes to run, as a function of the size of the input.
   - For AO*, the time complexity depends on the branching factor ((b)) and the maximum depth ((d)) or number of levels in the search tree.
   - Specifically, the time complexity of AO* is (O(b^d)).
2. Space Complexity:
   - Space complexity refers to the amount of memory an algorithm needs to run to completion.
   - In AO*, the space complexity is polynomial, which means it grows at most polynomial with the input size.
   - The space complexity is influenced by the AND feature in AO*, which reduces memory demand compared to other algorithms.
3. Optimality and Completeness:
   - AO* is not optimal because it stops as soon as it finds a solution. It doesn't explore all possible paths.
   - Unlike A*, which guarantees optimality, AO* doesn't guarantee the optimal solution.
   - However, AO* is complete in the sense that it will eventually find a solution if one exists.

Interface College of Computer Applications (ICCA)

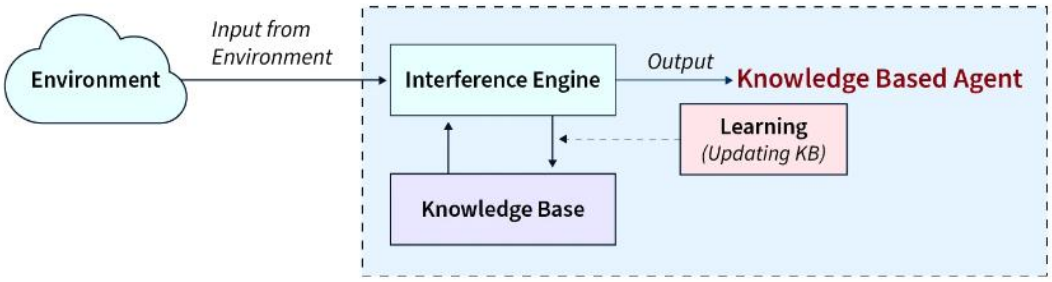
Unit III: - Knowledge Representation

Knowledge-Based Agents

- A knowledge-based agent is a type of intelligent agent in artificial intelligence that uses an internal knowledge base to make decisions and take actions.

The core components of a knowledge-based agent are:

- Knowledge Base (KB): A collection of sentences or facts about the world, typically represented in a formal language.
- Inference Engine: A set of procedures that the agent uses to derive new information or make decisions based on the information stored in the KB.



Structure of Knowledge Based Agents

Key operations performed by a knowledge-based agent on knowledge base are:

- Tell: Adding new information to the KB.
- Ask: Querying the KB to obtain information or infer new facts.

Working of Knowledge based agents

1. Perception: The agent perceives its environment through sensors.
2. Updating KB: The agent updates its KB with new perceptions.
3. Inference: The agent uses the inference engine to deduce new information from the KB.
4. Action: The agent decides on an action based on the inferred information and performs it using actuators.

Interface College of Computer Applications (ICCA)

Example: - Imagine the agent is like a robot in a house, trying to figure out what to do next:

1. Perception: - The robot sees or senses something in its environment (like seeing a chair or sensing a smell).
2. Updating KB: - It writes down this new information in its notebook using TELL.
3. Decision Making: - It asks its notebook (using ASK) what action to take based on what it knows. This involves reasoning about its current state and possible actions.
4. Action: - It decides on an action, writes down what it decided in its notebook (another TELL), and then performs the action.

```
function KB-AGENT(percept) returns an action
  static: KB, a knowledge base
          t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  action ← ASK(KB, MAKE-ACTION- QUERY(^))
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action
```

Figure 7.1 A generic knowledge-based agent.

Wumpus World

- Wumpus World is a well-known example used to illustrate knowledge-based agents and logical reasoning.
- It is a grid-based environment where an agent must navigate to achieve specific goals.
- The standard version consists of a 4x4 grid with the following elements:
  - Agent: The player or entity navigating the grid.
  - Wumpus: A monster that kills the agent if they enter its cell.
  - Pits: Deadly traps that kill the agent if they fall into them.
  - Gold: The treasure the agent aims to find and retrieve.
  - Breeze: An indicator that there is a pit in an adjacent cell.
  - Stench: An indicator that the Wumpus is in an adjacent cell.
  - Glitter: An indicator that gold is in the same cell as the agent.

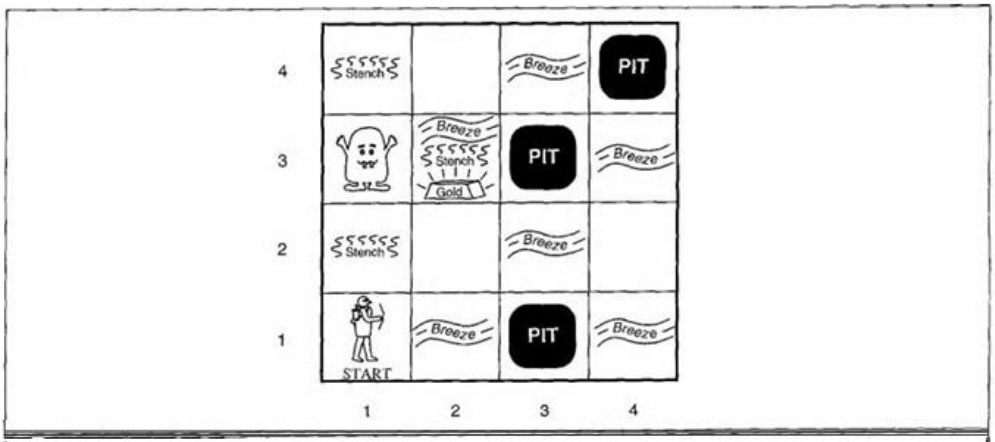


Figure 7.2 A typical wumpus world. The agent is in the bottom left corner.

(ICCA)

PEAS Description for the Wumpus World problem:

Performance measures:

- Agent gets the gold and return back safe = +1000 points
- Agent dies = -1000 points
- Each move of the agent = -1 point
- Agent uses the arrow = -10 points

Environment:

- A cave with 16(4×4) rooms

- Rooms adjacent (not diagonally) to the Wumpus are stinking
- Rooms adjacent (not diagonally) to the pit are breezy
- The room with the gold glitters
- Agent's initial position – Room[1, 1] and facing right side
- Location of Wumpus, gold and 3 pits can be anywhere, except in Room[1, 1].

Actuators:

- Devices that allow the agent to perform the following actions in the environment.
  - Move forward
  - Turn right
  - Turn left
  - Shoot(arrow)
  - Grab

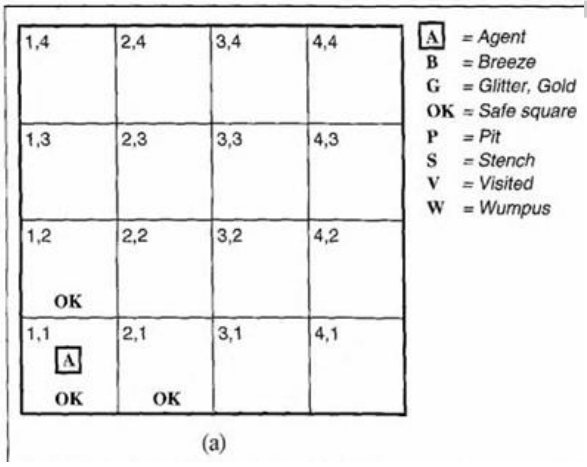
Sensors:

- Devices which helps the agent in sensing the following from the environment.
  - Breeze
  - Stench
  - Glitter
  - Scream (When the Wumpus is killed)
  - Bump (when the agent hits a wall)

Solution

- The agent receives percepts in the form of a list of five symbols [Stench,Breeze,gliter,bump,scream] indicating the presence or absence of various phenomena in its current location.
- For example: [Stench, Breeze, None, None, None] means there is a stench and a breeze but no glitter, bump, or scream.

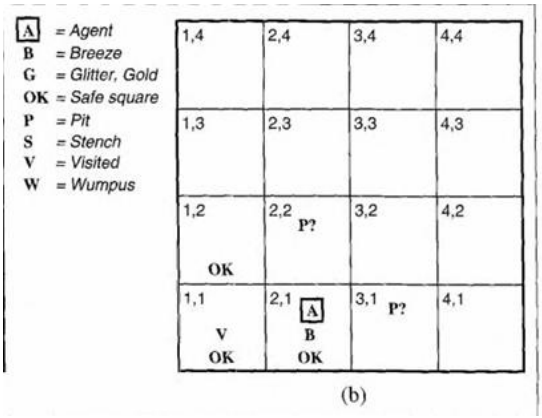
1. Initial State:
   - The agent starts at (1,1).
   - Initial percept: [None, None, None, None, None] (no stench, breeze, glitter, bump, or scream).
   - Conclusion: Neighboring cells (1,2) and (2,1) are safe (OK).



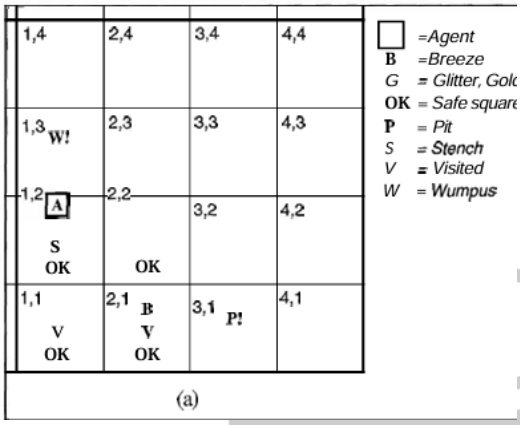
(a)

2. Move to (2,1):
   - The agent moves to cell (2,1).
   - Percept: [None, Breeze, None, None, None] (breeze detected, indicating a pit in an adjacent cell).
   - Conclusion: There might be a pit in (2,2) or (3,1) or both.

Interface Computer Applications (ICCA)



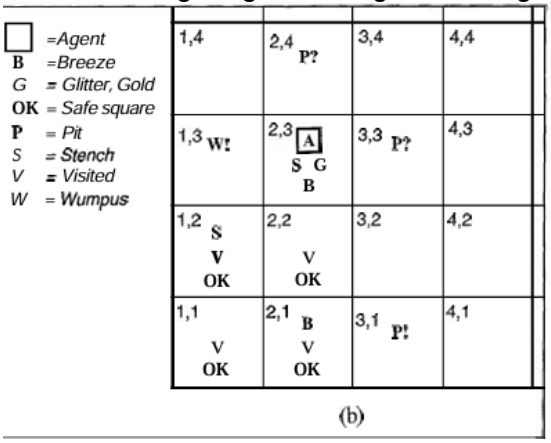
3. Move to (1,2):
   - The agent moves back to (1,1) and then to (1,2).
   - Percept: [Stench, None, None, None, None] (stench detected, indicating the Wumpus is nearby).
   - Conclusion: The Wumpus must be in (1,3) because it cannot be in (1,1) or (2,2). Also, no breeze in (1,2) means (2,2) is safe, so the pit must be in (3,1).



4. Move to (2,2):
   - The agent can safely move to (2,2) knowing it is safe.

Interface College of Computer Applications (ICCA)

5. Move to (2,3):
   - The agent moves to (2,3).
   - Percept: [None, None, Glitter, None, None] (glitter detected, indicating the gold is here).
   - Action: The agent grabs the gold, ending the game.



Note:- In each case where the agent draws a conclusion from the available information, that conclusion is guaranteed to be correct if the available information is correct. This is a fundamental property of logical reasoning.

Knowledge Representation

- Knowledge representation (KR) is a crucial aspect of artificial intelligence (AI), enabling systems to mimic human understanding and reasoning.
- In Artificial Intelligence (AI), knowledge representation (KR) refers to the methods and formalisms used to represent information, facts, rules, and other forms of knowledge in a way that allows intelligent systems to reason about and make inferences from that knowledge.
- It involves structuring information so that AI systems can use it to solve complex problems.

Some of the techniques of Knowledge Representation

1. Logical Representation
   - Logical representation uses formal logic to represent knowledge. It includes propositional logic and first-order logic (predicate logic).
   - Logical representation can be categorised into mainly two logics:
     - Propositional Logics
     - Predicate logics (First-Order Logic)
2. Semantic Networks
   - Semantic networks represent knowledge as a graph of nodes (concepts) and edges (relationships). They are used to illustrate how different concepts are related.
3. Frames
   - Frames are data structures for representing stereotypical situations. A frame has slots (attributes) and fillers (values).
   - A frame for a "Car" might include slots like "Make," "Model," "Year," and "Owner."
4. Rule-Based Systems
   - Rule-based systems represent knowledge in the form of rules (IF-THEN statements). They are widely used in expert systems.
   - Example: - IF (at bus stop AND bus arrives) THEN action (get into the bus)
5. Natural Language Processing (NLP)
   - NLP techniques are used to represent and process knowledge expressed in natural language.

Interface College of Computer Applications (ICCA)

Note: Do not be confused with logical representation and logical reasoning as logical representation is a representation language and reasoning is a process of thinking logically.

Logical representation and reasoning

- Syntax, semantics, models and Entailment are the foundational rules and concepts for logical representation techniques in AI. They provide a structured framework for representing and reasoning about knowledge logically..

a. Syntax
   - Syntax refers to the rules that define the structure of well-formed sentences in a language.
   - Example: In arithmetic, "x + y = 4" is syntactically correct, while "x2y+ =" is not.
   - Logical Syntax: In logical languages, sentences are constructed following specific syntactical rules. These sentences are stored in the agent's knowledge base (KB).

b. Semantics
   - Semantics deals with the meaning of sentences and their truth values in different possible worlds.
   - Truth and Possible Worlds: The truth of a sentence depends on the specific values assigned to variables in a model (a mathematical abstraction of a possible world).
   - Example: The sentence "x + y = 4" is true if x and y are assigned values such that their sum is 4.

   Models
   - A model (or possible world) is a mathematical abstraction that assigns truth or falsehood to each sentence.
   - Example: If x = 2 and y = 2, then the sentence "x + y = 4" is true in this model.

c. Entailment
   - In logical reasoning, entailment refers to the relationship between statements such that if one statement is true, then another statement must also be true.
   - A sentence α entails another sentence β if, in every model where α is true, β is also true.
   - Notation: We write α⊨β to indicate that α entails β.
   - Example: The sentence "x + y = 4" entails "4 = x + y" because in any model where "x + y = 4" is true, "4 = x + y" is also true.

d. Logical inference
   - Logical inference is the process of drawing conclusions or deriving new sentences from an existing knowledge base, such that the conclusions necessarily follow from the statements in KB according to the rules of logical entailment.
   - More precisely, given a knowledge base KB, which is a set of sentences representing the available knowledge, the goal of logical inference is to find a sentence α such that KB logically entails α, written as:

     KB |= α
   - The entailment relation |= is defined semantically based on models or possible worlds:
   - KB |= α if and only if in every model/possible world where all the sentences in KB are true, the sentence α is also true.

Propositional Logic

- In propositional logic, a proposition is a declarative statement that can be clearly determined to be either true or false.
- Propositional logic is a simple yet powerful form of logic used in various applications such as reasoning and problem-solving.
- It involves the use of propositions, logical connectives, and inference rules to form sentences and determine their truth values.

Syntax of Propositional Logic

- The syntax defines the structure of sentences in propositional logic, specifying which sentences are considered valid.

Atomic Sentences/Atomic proposition

- Atomic sentences are the simplest units in propositional logic.
- They consist of single proposition symbols, which represent propositions that can be true or false.
- Symbols are typically denoted by uppercase letters such as P, Q, R, etc.
- Example 1: - 2+2 is 4, it is an atomic proposition as it is a true fact.
  "The Sun is cold" is also a proposition as it is a false fact.
- Example 2: - $W_{1,3}$ is an atomic sentence representing the proposition "the wumpus is in location [1,3].

Interface College of Computer Applications (ICCA)

- W is a symbol that stands for the proposition about the wumpus.
- 1 and 3 are specific coordinates on a grid or map.
- $W_{1,3}$ as a whole is a single symbol representing a specific proposition: "the wumpus is in location [1,3].

Complex Sentences/ Complex proposition

- Complex sentences are formed by combining simpler sentences using logical connectives.
- Example :- "It is raining today, and street is wet."
  "Ankit is a doctor, and his clinic is in Mumbai."

Logical Connectives:

- Negation (¬): Represents "not". The negation of a proposition P is written as ¬P.
  - Example: $\neg W_{1,3}$. This proposition specifies that, “The Wumpus is not in location[1,3]”.
- Conjunction (∧): Represents "and". A conjunction of two propositions P and Q is written as P∧Q.
  - Example: $W_{1,3} \wedge P_{3,1}$, This proposition specifies that, "The wumpus is in location [1,3] and there is a pit in location [3,1]."
- Disjunction (∨): Represents "or". A disjunction of two propositions P and Q is written as P∨Q
  - Example: $(W_{1,3} \wedge P_{3,1}) \vee W_{2,2}$. This proposition specifies that, "(The wumpus is in location [1,3] and there is a pit in location [3,1]) or the wumpus is in location [2,2]."
- Implication (→): Represents "implies". An implication from P to Q is written as P→Q
  - Example: $(W_{1,3} \wedge P_{3,1}) \rightarrow \neg W_{2,2}$ This proposition specifies that, (if the wumpus is in [1,3] and the pit is in [3,1], then the wumpus is not in [2,2])
- Biconditional (↔): Represents "if and only if". A biconditional between P and Q is written as P↔Q.
  - Example: $W_{1,3} \leftrightarrow \neg W_{2,2}$, This proposition specifies that, (the wumpus is in [1,3] if and only if the wumpus is not in [2,2])

Sentence	→	*AtomicSentence* | *ComplexSentence*
AtomicSentence	→	**True** | **False** | Symbol
Symbol	→	*P* | **Q** | R | ...
ComplexSentence	→	¬ *Sentence*
	|	(*Sentence* ∧ *Sentence*)
	|	(*Sentence* ∨ *Sentence*)
	|	(*Sentence* ⇒ *Sentence*)
	|	(*Sentence* ⇔ *Sentence*)

Figure 7.7 A BNF (Backus–Naur Form) grammar of sentences in propositional logic.

Semantic for Propositional logic

- The semantics of propositional logic defines the rules for determining the truth value of a sentence within a given model.
- **A model** is an assignment of truth values (true or false) to each proposition symbol used in the sentences.
- For example, consider a scenario with three proposition symbols:
  - $P_{1,2}$ (meaning "there is a pit in [1,2]"), $P_{2,2}$, and $P_{3,1}$.
  - One possible model is m1 = { $P_{1,2}$ = false, $P_{2,2}$ = false, $P_{3,1}$ = true}.
- The number of possible models for a set of n proposition symbols is 2^n.

- This is because each proposition symbol can have two possible truth values (true or false), and there are n proposition symbols, leading to 2^n possible combinations of truth value assignments.
- With three proposition symbols, there are 2^3 = 8 possible models. For example

1. $P_{1,2} = \text{false}, P_{2,2} = \text{false}, P_{3,1} = \text{false}$
2. $P_{1,2} = \text{false}, P_{2,2} = \text{false}, P_{3,1} = \text{true}$
3. $P_{1,2} = \text{false}, P_{2,2} = \text{true}, P_{3,1} = \text{false}$
4. $P_{1,2} = \text{false}, P_{2,2} = \text{true}, P_{3,1} = \text{true}$
5. $P_{1,2} = \text{true}, P_{2,2} = \text{false}, P_{3,1} = \text{false}$
6. $P_{1,2} = \text{true}, P_{2,2} = \text{false}, P_{3,1} = \text{true}$
7. $P_{1,2} = \text{true}, P_{2,2} = \text{true}, P_{3,1} = \text{false}$
8. $P_{1,2} = \text{true}, P_{2,2} = \text{true}, P_{3,1} = \text{true}$

Compute the truth value

- The semantics for propositional logic must specify how to compute the truth value of any sentence, given a model. This is done recursively.
- All sentences are constructed from atomic sentences and the five connectives; therefore, we need to specify how to compute the truth of atomic sentences and how to compute the truth of sentences formed with each of the five connectives.

Atomic Sentences

- True: The atomic sentence "True" is true in every model.
- False: The atomic sentence "False" is false in every model.
- Consider the model m1 with three proposition symbols $P_{1,2}$,$P_{2,2}$ and $P_{3,1}$ m1 could be:
  $P_{1,2}$=false
  $P_{2,2}$=false
  $P_{3,1}$=true
- For any proposition symbol, the truth value is specified directly in the model. For example, if $P_{1,2}$ is false in model m1, then $P_{1,2}$ is false.

Complex Sentences

- To compute the truth value of complex sentences, we use the rules for logical connectives.

Logical Connectives

- **Negation ($\neg$):** The negation of a sentence s is true if s is false, and false if s is true.
- **Conjunction ($\wedge$):** The conjunction $s \wedge t$ is true if both s and t are true, otherwise it is false.
- **Disjunction ($\vee$):** The disjunction $s \vee t$ is true if at least one of s or t is true, otherwise it is false
- **Implication ($\rightarrow$):** The implication $s \rightarrow t$ is false if s is true and t is false; otherwise, it is true.
- **Biconditional ($\leftrightarrow$):** The biconditional $s \leftrightarrow t$ is true if s and t have the same truth value (both true or both false).

Truth Table

Interface College of Computer Applications (ICCA)

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*	*false*
true	*false*	*false*	*false*	*true*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*

A simple knowledge base

Let's construct the knowledge base for the Wumpus World, focusing on pits and breezes.

1. Vocabulary
   - $P_{i,j}$: True if there is a pit in cell [i,j]
   - $B_{i,j}$: True if there is a breeze in cell [i,j].
2. Knowledge Base Sentences.
   - Sentence 1: No Pit in [1,1] R1: - $\neg P_{1,1}$
   - Sentence 2: Breezy Squares and Neighboring Pits
     - A square [i,j] is breezy if and only if there is a pit in one of its neighboring squares. We'll state this for a few relevant squares:
       a) For square [1,2] R2: $B_{1,1} \leftrightarrow (P_{1,2} \vee P_{2,1})$
       b) For square [2,1] R3: $B_{2,1} \leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$
   - Sentence 3: Percepts in the First Two Visited Squares
     a) There is no breeze in [1,1] R4: $\neg B_{1,1}$
     b) There is a breeze in [2,1] R5: $B_{2,1}$
3. Full Knowledge base

   KB = $(\neg P_{1,1}) \wedge (B_{1,1} \leftrightarrow (P_{1,1} \vee P_{2,1})) \wedge (B_{2,1} \leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1}))) \wedge \neg B_{1,1} \wedge B_{2,1}$

Logical Inference

- For propositional logic, models are assignments of true or false to every proposition symbol.
- From the above KB sentences, the relevant proposition symbols are $B_{1,1}, B_{2,1}, P_{1,1}, P_{1,2}, P_{2,1}, P_{2,2}$ and $P_{3,1}$.

  With seven symbols, there are 2^7 = 128 possible models; in three of these, KB is true

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	KB
false	*false*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*false*	*false*
false	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*false*	*true*	*false*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
false	*true*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*false*	*true*	*true*	*false*
false	*true*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*true*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
true	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*true*	*false*	*true*	*false*

α_1 = "There is no pit in [1,2]."

α_2 = "There is no pit in [2,2]."

In those three models, $\neg P_{1,2}$ is true, hence there is no pit in [1,2]. i.e KB |= α1
On the other hand, P2,~ is true in two of the three models and false in one, so we cannot yet tell whether there is a pit in [2,2]. KB |≠ α2

Propositional theorem proving

- Propositional theorem proving involves systematically deriving conclusions from a set of premises using formal inference rules. These rules allow us to manipulate and combine propositions to produce new propositions, ultimately leading to the desired conclusion.

Interface College of Computer Applications (ICCA)

Inference Rules

① Modus Ponens (MP)

$$\frac{P \rightarrow Q,\ P}{Q}$$

If $P \rightarrow Q$ and P are true, then Q is true.

② And - Elimination ($\wedge E$)

$$\frac{P \wedge Q}{P} \quad \text{and} \quad \frac{P \wedge Q}{Q}$$

From $P \wedge Q$, infer P and Q separately.

③ Biconditional Elimination ($\leftrightarrow E$)

$$P \leftrightarrow Q \equiv (P \rightarrow Q) \wedge (Q \rightarrow P)$$

From $P \leftrightarrow Q$, infer $(P \rightarrow Q) \wedge (Q \rightarrow P)$

④ Contrapositive (contr)

$$P \rightarrow Q \equiv \sim Q \rightarrow \sim P$$

If $P \rightarrow Q$ is true, then $\sim Q \rightarrow \sim P$ is also true.

⑤ De Morgan's Law (DM)

$$\sim (P \wedge Q) \equiv \sim P \vee \sim Q$$
$$\sim (P \vee Q) \equiv \sim P \wedge \sim Q$$

⑥ Disjunctive Syllogism (DS)

$$\frac{P \vee Q,\ \sim P}{Q}$$

From $P \vee Q$ and $\sim P$, infer Q

Example :- Consider a KB in the Wumpus World and a Proof involving several inference rules to demonstrate that there is no pit in [1,2]

Knowledge Base Sentences

R_1 : $\sim P_{1,1}$

R_2 : $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$

R_3 : $B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$

R_4 : $\sim B_{1,1}$

R_5 : $B_{2,1}$

Goal :- To prove $\sim P_{1,2}$

Step 1 :- Apply biconditional elimination to R_2.

R_6 : $(B_{1,1} \rightarrow P_{1,2} \vee P_{2,1}) \wedge (P_{1,2} \vee P_{2,1} \rightarrow B_{1,1})$

Step 2 :- Apply And elemination to R_6.

R_7 : $(P_{1,2} \vee P_{2,1} \rightarrow B_{1,1})$

Step 3 :- Logical equivalence for contrapositives

R_8 : $(\sim B_{11} \rightarrow \sim(P_{1,2} \vee P_{21}))$

Step 4 :- Apply Modus Ponens with R_8 and the Percept R_4

R_4 : $\sim B_{11}$

R_8, R_4

$(\sim B_{11} \rightarrow (\sim(P_{1,2} \vee P_{21})), \sim B_{11}$

$\therefore$ R_9 : $\sim(P_{12} \vee P_{21})$

Step 5 :- Apply DeMorgan's rule to R_9

R_{10} : $\sim P_{12} \wedge \sim P_{21}$.

That is neither [1,2] and [2,1] contains a pit.

Resolution-Based Inference Algorithm

- The resolution-based inference algorithm uses proof-by-contradiction to determine whether a given knowledge base (KB) entails a particular proposition α.

Here are the key steps:

1. Negate the Proposition: To prove that KB⊨α, we start by assuming ¬α and try to derive a contradiction.
2. Combine with Knowledge Base: Form a new set of clauses from the conjunction of the knowledge base KB and the negation of the proposition, i.e., KB ∧ ¬α.
3. Convert to CNF: Convert the combined knowledge base and negated proposition into Conjunctive Normal Form (CNF), which is a conjunction of disjunctions of literals (clauses).
4. Apply Resolution Rule: Use the resolution rule to iteratively generate new clauses by resolving pairs of existing clauses. The resolution rule states that for two clauses C1=(A∨L) and C2=(¬L∨B), their resolvent is the clause C=(A∨B), which removes the complementary literals L and ¬L.
5. Check for Empty Clause: Continue applying the resolution rule until either:
6. An empty clause (□) is produced, indicating a contradiction and thereby proving KB⊨α
7. there are no new clauses that can be added, in which case KB does not entail α.

Example :-

Interface College of Computer Applications (ICCA)

Step 1:- Knowledge Base Sentences

R_1 : $\sim P_{11}$

R_2 : $B_{11} \Leftrightarrow (P_{12} \vee P_{21})$

R_3 : $B_{21} \Leftrightarrow (P_{11} \vee P_{22} \vee P_{31})$

R_4 : $\sim B_{11}$

R_5 : B_{21}

Step 2:- Proposition $\alpha = \sim P_{12}$

Step 3:- Proof by Contradiction.

$KB \models -\alpha$

$\alpha = \sim P_{12}$

Convert to CNF

Apply Double Negation Rule

$\sim(\sim P_{12}) = P_{12}$

$\therefore \alpha = P_{12}$

Double Negation

$\sim(\sim A) \equiv A$

Step 4:- Combine KB and $-\alpha$.

$R_2 \wedge R_4 \wedge \alpha$.

$= B_{11} \Leftrightarrow (P_{12} \vee P_{21}) \wedge \sim B_{11} \wedge P_{12}$.

Step 5:- Convert to CNF.

① $B_{11} \Leftrightarrow (P_{12} \vee P_{21})$

Apply Bi-directional elimination

$B_{11} \rightarrow (P_{12} \vee P_{21})$ and $(P_{21} \vee P_{21}) \rightarrow B_{11}$

Apply Implication elimination.

$\sim B_{11} \vee (P_{12} \vee P_{21})$ and

$\sim (P_{12} \vee P_{21}) \vee B_{11}$

For the eq $\sim (P_{12} \vee P_{21}) \vee B_{11}$ Apply De Morgan law.

$(\sim P_{12} \wedge \sim P_{21}) \vee B_{11}$

Apply Distributive law

$(\sim P_{12} \vee B_{11}) \wedge (\sim P_{21} \vee B_{11})$

CNF is.

$\{\sim B_{11} \vee (P_{12} \vee P_{21}) \wedge (\sim P_{12} \vee B_{11}) \wedge (\sim P_{21} \vee B_{11}) \wedge P_{12}$.

Implication elimination.

$(\alpha \rightarrow \beta) \equiv (\neg \alpha \vee \beta)$

De Morgan law

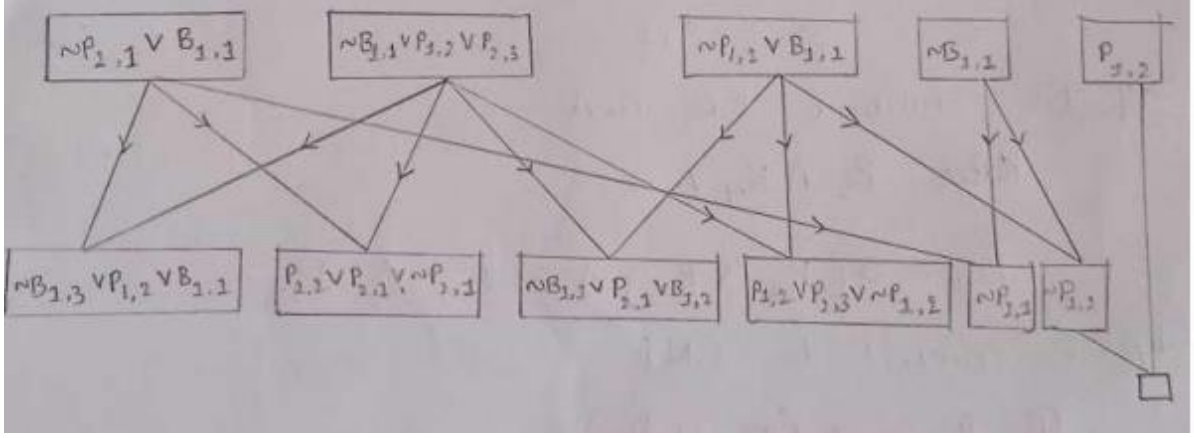
$\neg(\alpha \wedge \beta) \equiv (\neg \alpha \vee \neg \beta)$

$\neg(\alpha \vee \beta) \equiv (\neg \alpha \wedge \neg \beta)$

Distributive Law

$\alpha \wedge (\beta \vee \gamma) \equiv (\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$

$\alpha \vee (\beta \wedge \gamma) \equiv (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$



```
function PL-RESOLUTION(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
          α, the query, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of KB ∧ ¬α
  new ← { }
  loop do
    for each C_i, C_j in clauses do
      resolvents ← PL-RESOLVE(C_i, C_j)
      if resolvents contains the empty clause then return true
      new ← new ∪ resolvents
    if new ⊆ clauses then return false
    clauses ← clauses ∪ new
```

Figure 7.12 A simple resolution algorithm for propositional logic. The function PL-RESOLVE returns the set of all possible clauses obtained by resolving its two inputs.

Effective Prepositional Model Checking

- Two families of efficient algorithms for propositional inference based on model checking: one approach based on backtracking search, and one on hill climbing search. These algorithms are part of the "technology" of propositional logic.

Backtracking Algorithm

- The Davis-Putnam-Logemann-Loveland (DPLL) algorithm is a refined and efficient approach for solving the Boolean satisfiability problem (SAT), which is the task of determining if there exists an assignment of truth values to variables that makes a given Boolean formula true.
- The Davis-Putnam-Logemann-Loveland (DPLL) algorithm is an enhancement of the basic backtracking algorithm for solving the Boolean satisfiability problem (SAT). It builds on the basic idea of depth-first search but incorporates several heuristics to improve efficiency.
- This formula is typically given in Conjunctive Normal Form (CNF), consisting of a conjunction of clauses where each clause is a disjunction of literals.

Key Improvements in DPLL Algorithm

1. Early Termination:
   - True: If any clause in the formula is true with the current partial assignment, the entire formula might be judged as true.
   - False: If any clause is false under the current partial assignment (all its literals are false), the entire formula is false, prompting a backtrack.
2. Pure Symbol Heuristic:
   - A pure symbol is a symbol that appears with only one polarity (either always positive or always negative) in the remaining clauses.
   - Pure symbols can be assigned truth values that satisfy all clauses in which they appear without making any clause false. This reduces the search space.
3. Unit Clause Heuristic:
   - A unit clause is a clause that contains only one unassigned literal.
   - Unit propagation assigns values to these literals directly, often triggering a cascade of forced assignments. This process is similar to forward chaining in Horn clauses.

Steps in DPLL Algorithm

- Initialization: Start with the given set of clauses and symbols.
- Early Termination: Check if the current model satisfies all clauses or if any clause is unsatisfied.
- Pure Symbol Heuristic: Assign pure symbols if any.

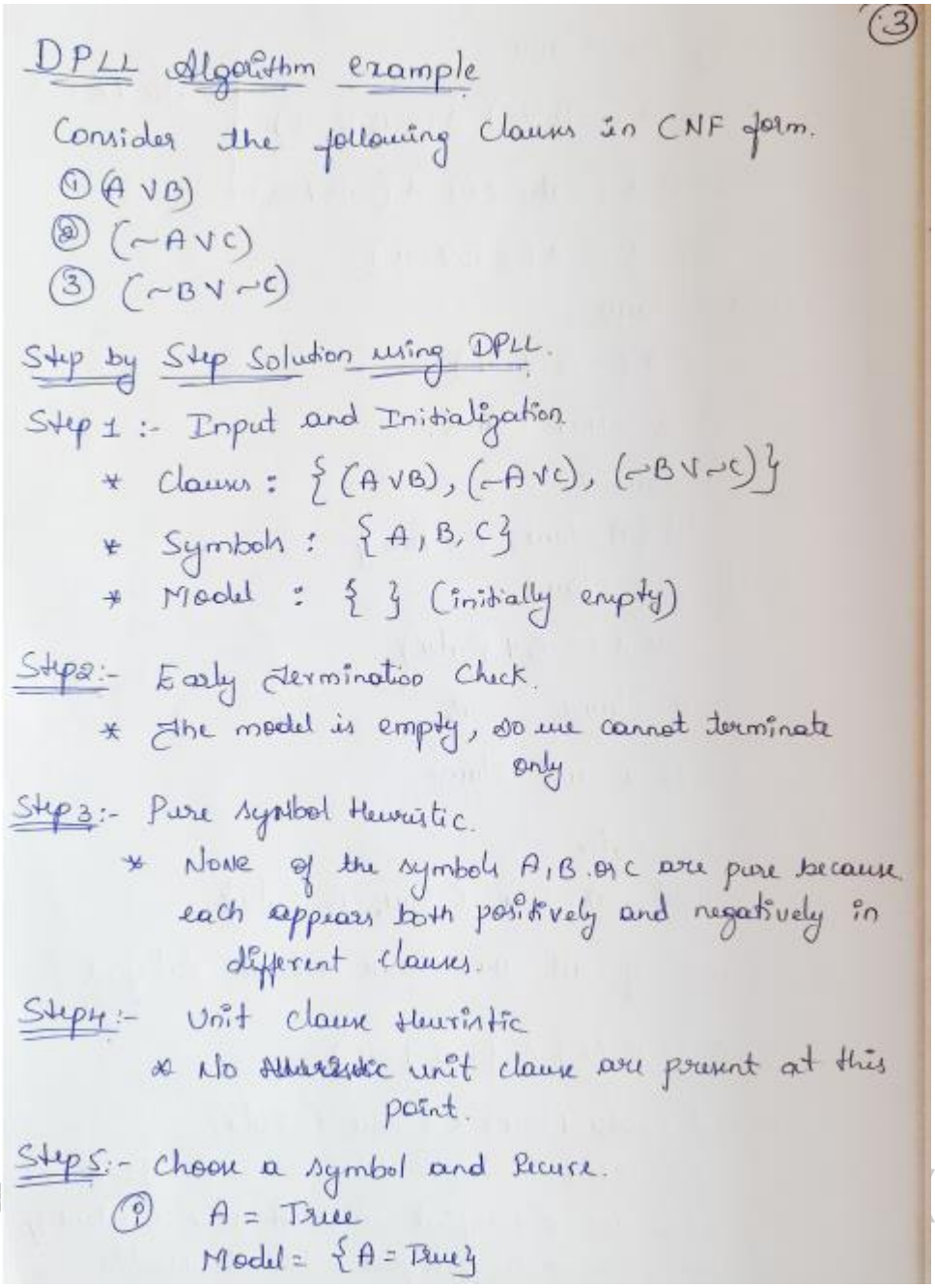
Interface College of Computer Applications (ICCA)

- Unit Clause Heuristic: Assign values to unit clauses and propagate.
- Recursive Search: Choose an unassigned symbol, try both assignments, and recursively apply DPLL.

```
function DPLL-SATISFIABLE?(s) returns true or false
  inputs: s, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of s
  symbols ← a list of the proposition symbols in s
  return DPLL(clauses, symbols, [])

function DPLL(clauses, symbols, model) returns true or false

  if every clause in clauses is true in model then return true
  if some clause in clauses is false in model then return false
  P, value ← FIND-PURE-SYMBOL(symbols, clauses, model)
  if P is non-null then return DPLL(clauses, symbols – P, EXTEND(P, value, model)
  P, value ← FIND-UNIT-CLAUSE(clauses, model)
  if P is non-null then return DPLL(clauses, symbols – P, EXTEND(P, value, model)
  P ← FIRST(symbols); rest ← REST(symbols)
  return DPLL(clauses, rest, EXTEND(P, true, model)) or
         DPLL(clauses, rest, EXTEND(P, false, model))
```

Example

ICCA
Interface College of Computer Applications (ICCA)



(3)

DPLL Algorithm example

Consider the following clauses in CNF form.

① (A ∨ B)

② (¬A ∨ C)

③ (¬B ∨ ¬C)

Step by Step Solution using DPLL.

Step 1 :- Input and Initialization

* Clauses : {(A ∨ B), (¬A ∨ C), (¬B ∨ ¬C)}
* Symbols : {A, B, C}
* Model : { } (initially empty)

Step 2 :- Early Termination Check.

* The model is empty, so we cannot terminate only

Step 3 :- Pure Symbol Heuristic.

* None of the symbols A, B, & C are pure because each appears both positively and negatively in different clauses.

Step 4 :- Unit clause Heuristic

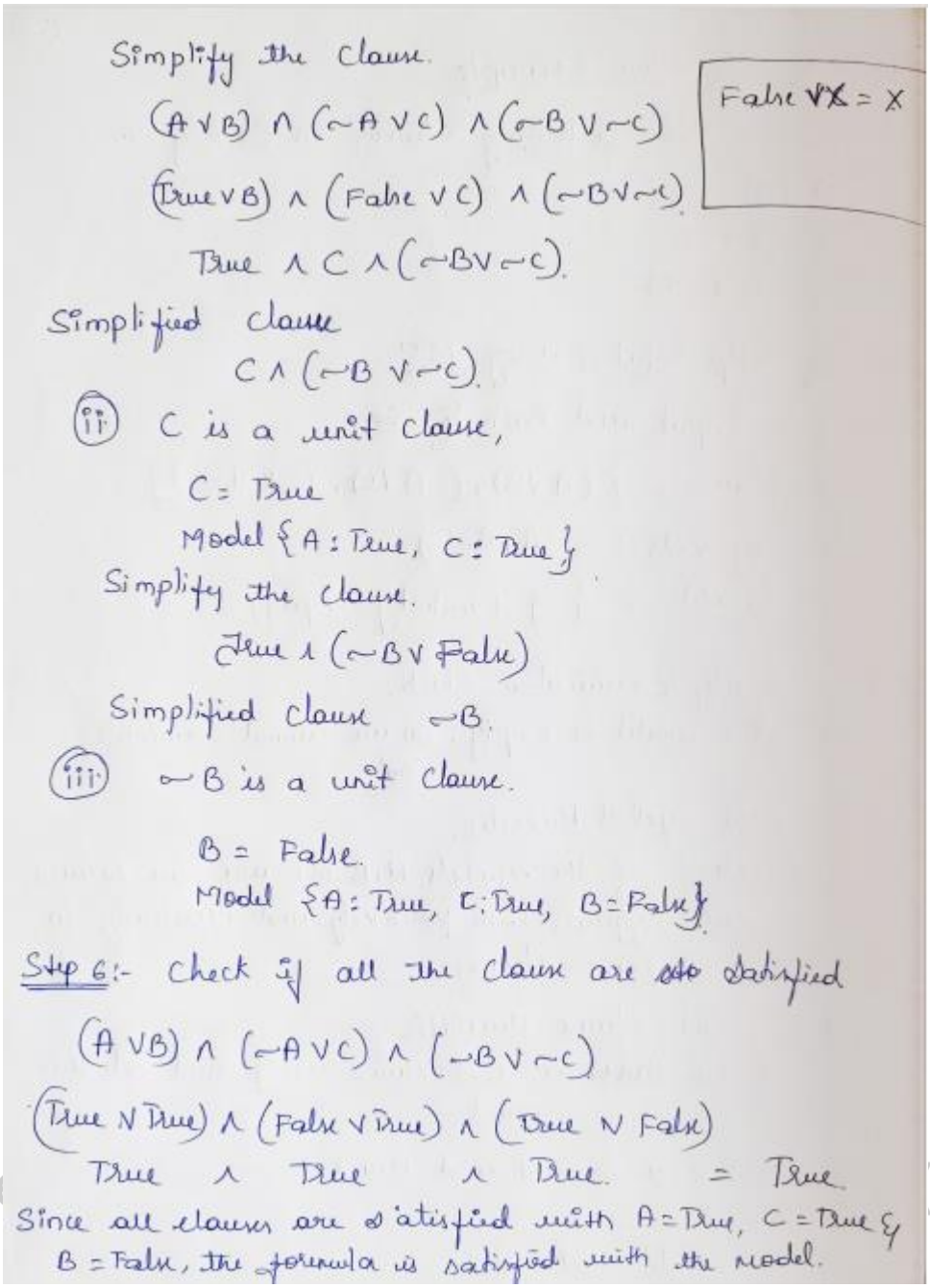
* No ~~Heuristic~~ unit clause are present at this point.

Step 5 :- Choose a symbol and Recurse.

① A = True

Model = {A = True}

Int (ICCA)



Simplify the clause.

$(A \vee B) \wedge (\neg A \vee C) \wedge (\neg B \vee \neg C)$

$(True \vee B) \wedge (False \vee C) \wedge (\neg B \vee \neg C)$

$True \wedge C \wedge (\neg B \vee \neg C)$

False ∨ X = X

Simplified clause

$C \wedge (\neg B \vee \neg C)$

(ii) C is a unit clause,

C = True

Model {A: True, C: True}

Simplify the clause.

$True \wedge (\neg B \vee False)$

Simplified clause $\neg B$.

(iii) $\neg B$ is a unit clause.

B = False.

Model {A: True, C: True, B = False}

Step 6:- Check if all the clause are ~~sto~~ satisfied

$(A \vee B) \wedge (\neg A \vee C) \wedge (\neg B \vee \neg C)$

$(True \vee True) \wedge (False \vee True) \wedge (True \vee False)$

$True \wedge True \wedge True = True$

Since all clause are satisfied with A = True, C = True & B = False, the formula is satisfied with the model.

Int (ICCA)

Local Search Algorithm

- Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem.
- It terminates when it reaches a peak value where no neighbor has a higher value.
- Hill climbing algorithm is a technique which is used for optimizing the mathematical problems. One of the widely discussed examples of Hill climbing algorithm is Traveling-salesman Problem in which we need to minimize the distance traveled by the salesman.
- It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that.
- A node of hill climbing algorithm has two components which are state and value.
- Hill Climbing is mostly used when a good heuristic is available.

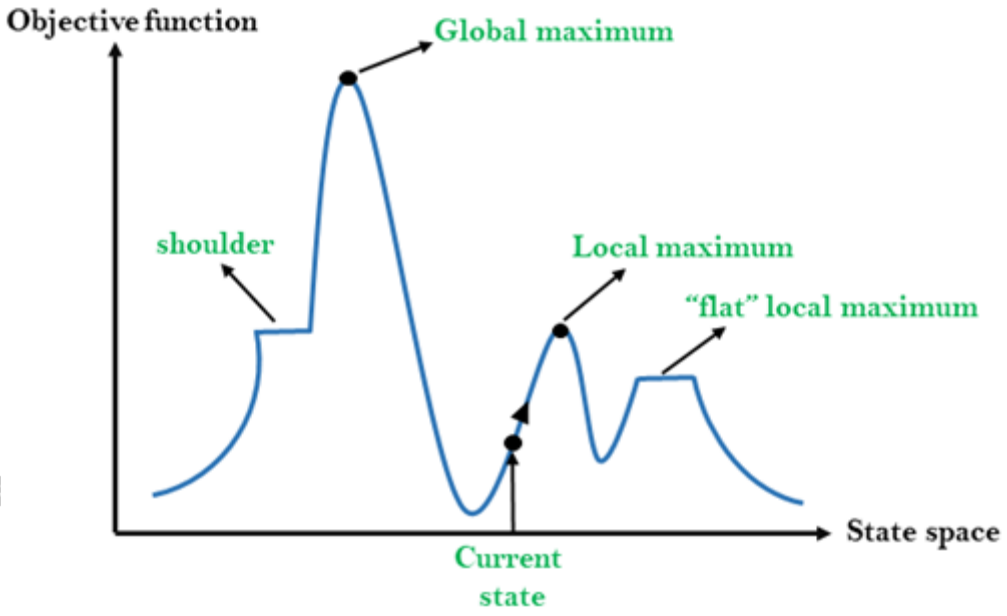
- In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

Features of Hill Climbing:

- Generate and Test variant: Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.
- Greedy approach: Hill-climbing algorithm search moves in the direction which optimizes the cost.
- No backtracking: It does not backtrack the search space, as it does not remember the previous states.

State-space Diagram for Hill Climbing:

- The state-space landscape is a graphical representation of the hill-climbing algorithm which is showing a graph between various states of algorithm and Objective function/Cost.
- On Y-axis we have taken the function which can be an objective function or cost function, and state-space on the x-axis. If the function on Y-axis is cost then, the goal of search is to find the global minimum and local minimum. If the function of Y-axis is Objective function, then the goal of the search is to find the global maximum and local maximum.



Different regions in the state space landscape:

- Local Maximum: Local maximum is a state which is better than its neighbor states, but there is also another state which is higher than it.
- Global Maximum: Global maximum is the best possible state of state space landscape. It has the highest value of objective function.
- Current state: It is a state in a landscape diagram where an agent is currently present.
- Flat local maximum: It is a flat space in the landscape where all the neighbor states of current states have the same value.
- Shoulder: It is a plateau region which has an uphill edge.

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  inputs: problem, a problem
  local variables: current, a node
                   neighbor, a node

  current ← MAKE-NODE(INITIAL-STATE[problem])
  loop do
     neighbor ← a highest-valued successor of current
     if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
     current ← neighbor
```

Types of Hill Climbing Algorithm:

- Simple hill Climbing:
- Steepest-Ascent hill-climbing:
- Stochastic hill Climbing:

1. Simple Hill Climbing:

- Simple hill climbing is the simplest way to implement a hill climbing algorithm. It only evaluates the neighbor node state at a time and selects the first one which optimizes current cost and set it as a current state. It only checks it's one successor state, and if it finds better than the current state, then move else be in the same state. This algorithm has the following features:

2. Steepest-Ascent hill climbing:

- The steepest-Ascent algorithm is a variation of simple hill climbing algorithm. This algorithm examines all the neighboring nodes of the current state and selects one neighbor node which is closest to the goal state. This algorithm consumes more time as it searches for multiple neighbors

3. Stochastic hill climbing:

- Stochastic hill climbing does not examine for all its neighbor before moving. Rather, this search algorithm selects one neighbor node at random and decides whether to choose it as a current state or examine another state.

Agent Based on Prepositional logic

Two approaches for creating agents that use propositional logic in the context of the Wumpus World game:

Interface College of Computer Applications (ICCA)

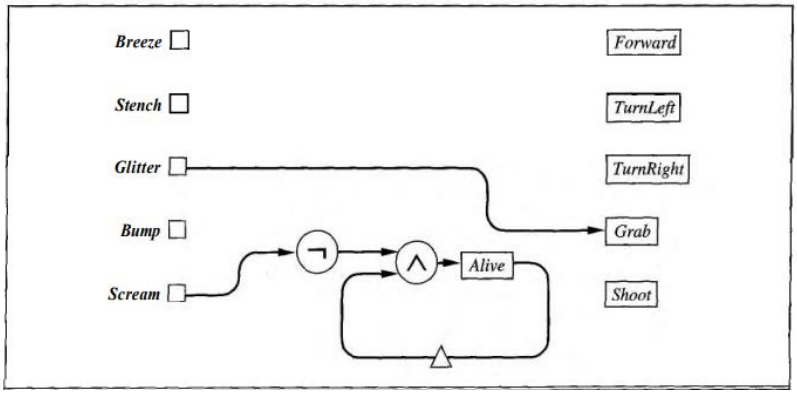
1. Inference Based Agents

- Basic Structure:
  - Inference-based agents are a type of knowledge-based agent
  - They consist of two main components: a) A knowledge base (KB) b) An inference mechanism
- Knowledge Base (KB):
  - Repository of facts and rules about the world
  - In the Wumpus World example:
  - Contains "physics" of the world
  - Includes initial facts (e.g., [1,1] is safe: ¬P1,1 ∧ ¬W1,1)
  - Contains rules about environment (e.g., how breezes and stenches arise)
  - May include constraints (e.g., exactly one Wumpus exists)
- Knowledge Representation:
  - Uses propositional logic in this case
  - Each proposition represents a fact about the world
  - Complex statements formed using logical connectives (∧, ∨, ¬, ⇒, ⇔)

- Inference Mechanism:
  - Algorithms used to derive new information from the KB
  - Common methods include: DPLL (Davis–Putnam–Logemann–Loveland) ,WALKSAT, Resolution
  - These algorithms perform logical operations to determine entailment
- Agent Operation:
  a) Perception: Agent receives percepts from the environment
  b) KB Update: New percepts are added to the KB
  c) Query: Agent formulates a query about the safety of potential moves
  d) Inference: Uses the inference mechanism to determine if the move is safe
  e) Action: Chooses and executes the safest available move
- Reasoning Process:
  - Uses entailment to prove safety of squares
  - A square [i,j] is provably safe if KB $\models (\neg P_{i,j} \wedge \neg W_{i,j})$
  - If safety can't be proved, it looks for possibly safe squares
  - Possibly safe: KB $\not\models (P_{i,j} \vee W_{i,j})$
- Advantages:
  a) Flexibility: Easy to add new rules or knowledge
  b) Completeness: Can perform complete logical inference
  c) Explainability: Reasoning process can be traced and explained
  d) Generality: Can handle a wide range of problem types

2. Circuit Based agent

- Basic Structure:
  - Circuit-based agents are a type of reflex agent with state
  - Implemented as networks of logical gates and registers



Interfac ns (ICCA)

- Components:
  a) Gates: Implement logical connectives (AND, OR, NOT, etc.)
  b) Registers: Store truth values of propositions
  c) Inputs: Receive percepts from the environment
  d) Outputs: Action registers (e.g., Move, Grab, Shoot)
  e) Delay Lines: Maintain state across time steps
- Knowledge Representation:
  - Truth values stored directly in registers
  - Logical relationships encoded in the circuit structure
  - Each register typically represents a proposition about the world
- Operation Principle:
  - Signals propagate through the circuit in a dataflow fashion
  - At each time step:
    a) Inputs are set based on percepts
    b) Signals flow through gates

c) Registers update their values
d) Output registers determine agent's actions

- Handling Time and State:
  - Registers store current truth values
  - Delay lines feed register outputs back into the circuit
  - Example: Alive register for Wumpus state Alivet ↔ ¬Screamt ∧ Alivet-1
- Handling Location:
  - Separate register for each possible location (e.g., L1,1)
  - Complex circuits update location based on moves and percepts
- Advantages:
  a) Efficient: Fast operation once the circuit is designed
  b) Natural time handling: Uses registers and delay lines
  c) Compact representation: No need for explicit time indexing of propositions
  d) Parallelism: Different parts of the circuit can operate simultaneously

First Order Logic

- First Order Logic in Artificial Intelligence is a technique used for knowledge representation.
- It is an extension of propositional logic and unlike propositional logic, it is sufficiently expressive in representing any natural language construct.
- First Order Logic in AI is also known as Predicate Logic or First Order Predicate Logic.
- First Order Logic in Artificial Intelligence doesn't only include facts but also different other entities as listed below.
  - Objects: - Objects can denote any real-world entity or any variable. E.g., A, B, colours, theories, circles etc.
  - Relations: -Relations represent the links between different objects. Relations can be unary(relations defined for a single term) and n-ary(relations defined for n terms). E.g., blue, round (unary); friends, siblings (binary); etc.
  - Functions: - Functions map their input object to the output object using their underlying relation. Eg: father_of(), mother_of() etc.
- First-order logic in Artificial Intelligence comprises two main components, which are as follows.
  - Syntax:- Syntax represents the rules to write expressions in First Order Logic in Artificial Intelligence.
  - Semantics:- Semantics refers to the techniques that we use to evaluate an expression of First Order Logic in AI. These techniques use various known relations and facts of the respective environment to deduce the boolean value of the given First Order Logic expression.

Interface College of Computer Applications (ICCA)

Syntax of First Order Logic

- The syntax of FOL determines which collection of symbols is a logical expression in first-order logic.
- The basic syntactic elements of first-order logic are symbols.
- We write statements in short-hand notation in FOL.
- Following are the basic elements of FOL syntax:

Constant	1, 2, A, John, Mumbai, cat,....
Variables	x, y, z, a, b,....
Predicates	Brother, Father, >,....
Function	sqrt, LeftLegOf, ...
Connectives	∧, ∨, ¬, ⇒, ⇔
Equality	==
Quantifier	∀, ∃

Atomic Sentence

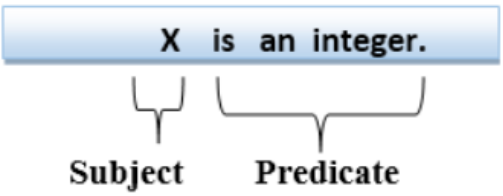
- Atomic sentences are the most basic sentences of first-order logic.
- These sentences are formed from a predicate symbol followed by a parenthesis with a sequence of terms.
- We can represent atomic sentences as Predicate (term1, term2, ..., term n).
- Example: Ravi and Ajay are brothers: => Brothers(Ravi, Ajay).
  Chinky is a cat: => cat (Chinky).

Complex Sentences:

- Complex sentences are made by combining atomic sentences using connectives.
- First-order logic statements can be divided into two parts:
  - Subject: Subject is the main part of the statement.
  - Predicate: A predicate can be defined as a relation, which binds two atoms together in a statement.

- Consider the statement: "x is an integer.", it consists of two parts, the first part x is the subject of the statement and second part "is an integer," is known as a predicate.

Interface College of Computer Applications (ICCA)



Quantifiers

- Quantifiers in First Order Logic in AI, as the name suggests, are used to quantify any entity in a given environment.
- Quantification refers to the identification of the total number of an entity that is present in the environment and satisfies a given expression in First Order Logic in Artificial Intelligence.
- Quantifiers enable us to determine the range and scope of a variable in a logical expression.
- Two types of quantifiers are stated as follows.

- Universal Quantifier
- Existential Quantifier

Universal Quantifier

- Universal Quantifier in First Order Logic in AI is a symbol in a logical expression that signifies that the given expression is true in its range for all instances of the concerned entity.
- It is represented by the symbol ∀(an inverted A). If x is a variable, then x is read as "For all x" or "For every x" or "For each x".
- For example, let us take the sentence, "All cats like fish". Let us take a variable x which can take the value of "cat". Let us take a predicate cat (x) which is true if x is a cat.
- Similarly, let us take another predicate likes (x,y) which is true if x likes y.
- Therefore, using the universal quantifier ∀, we can write

  ∀x cat (x)⟹ likes (x, fish).
- This expression is read as "For all x, if x is a cat, then x likes to fish".

Existential Quantifier

- An existential Quantifier in First Order Logic in Artificial Intelligence is a symbol in a logical expression that signifies that the given expression is true in its range for at least one of the instances of the concerned entity.
- It is represented by the symbol ∃ (an inverted E). If x is a variable, then ∃x is read as "There exists x" or "For some x" or "For at least one x".
- For example, let us take the sentence, "Some students like ice cream". Let us take a variable x which can take the value of "student". Let us take a predicate student (x), which is true if x is a student. Similarly, let us take another predicate likes (x,y), which is true if x likes y. Therefore, using the existential quantifier ∃, we can write

  ∃x student (x)∧ likes (x, ice-cream)
- This expression reads, "There exists some xx such that xx is a student and also likes ice cream".

Using First Order Logic

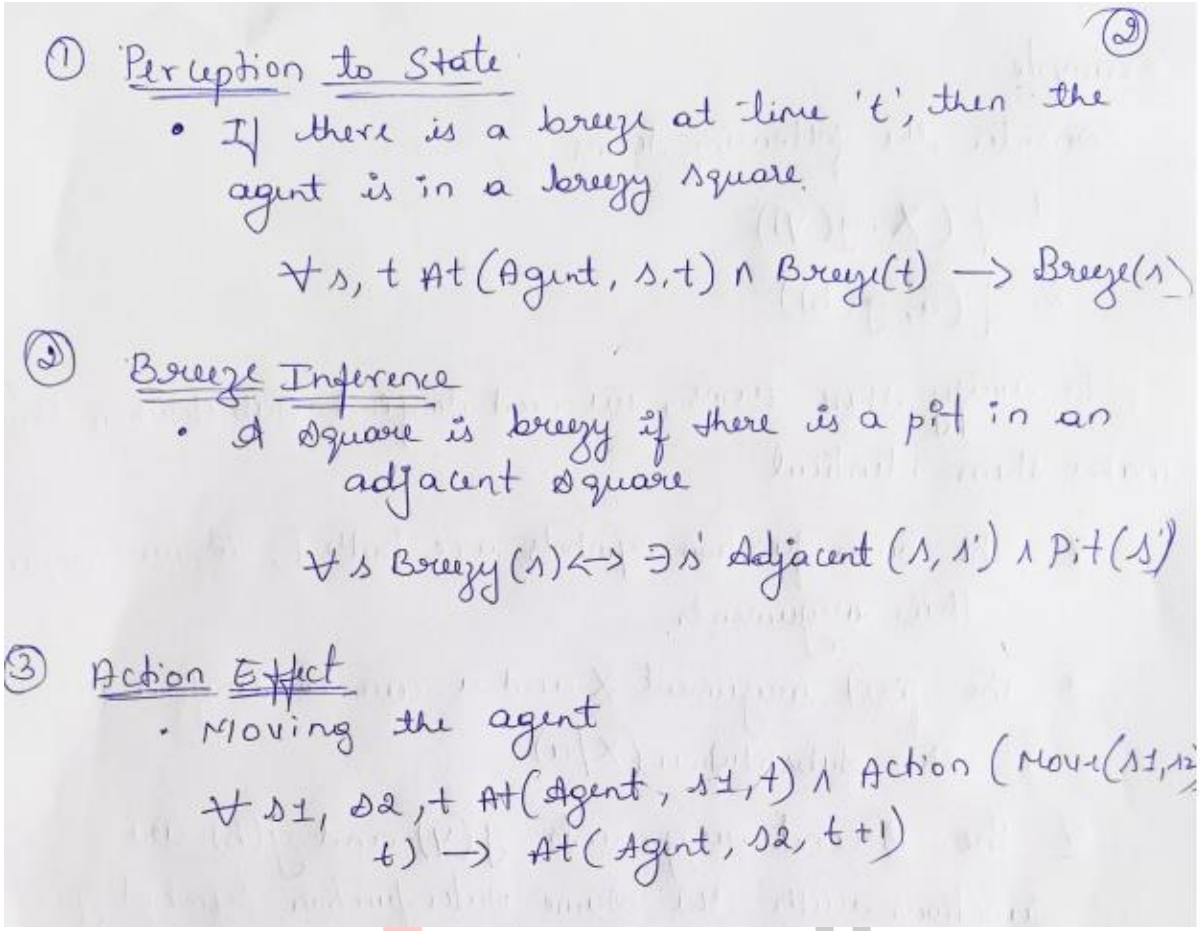
Interface College of Computer Applications (ICCA)

Using First order Logic.

Representing Wumpus World using First-Order Logic.

Wumpus World

* The Wumpus world consists of a grid of squares. The agent can move from square to square, but some squares contain hazards.
  - Pits : If the agent moves into a square with a pit, it falls in and dies.
  - Wumpus : A monster that can kill the agent. The Wumpus can be shot with an arrow.
* The agent perceives the world through certain sensory inputs.
  - Stench : Detected in squares adjacent to the Wumpus.
  - Breeze : Detected in squares adjacent to a pit.
  - Glitter : Detected in the square where the gold is located.
  - Bump : Detected when the agent walks into a wall.
  - Scream : Heard when the Wumpus is killed.



① Perception to State

- If there is a breeze at time 't', then the agent is in a breezy square.

$\forall s, t \; At(Agent, s, t) \wedge Breezy(t) \rightarrow Breeze(s)$

② Breeze Inference

- A square is breezy if there is a pit in an adjacent square

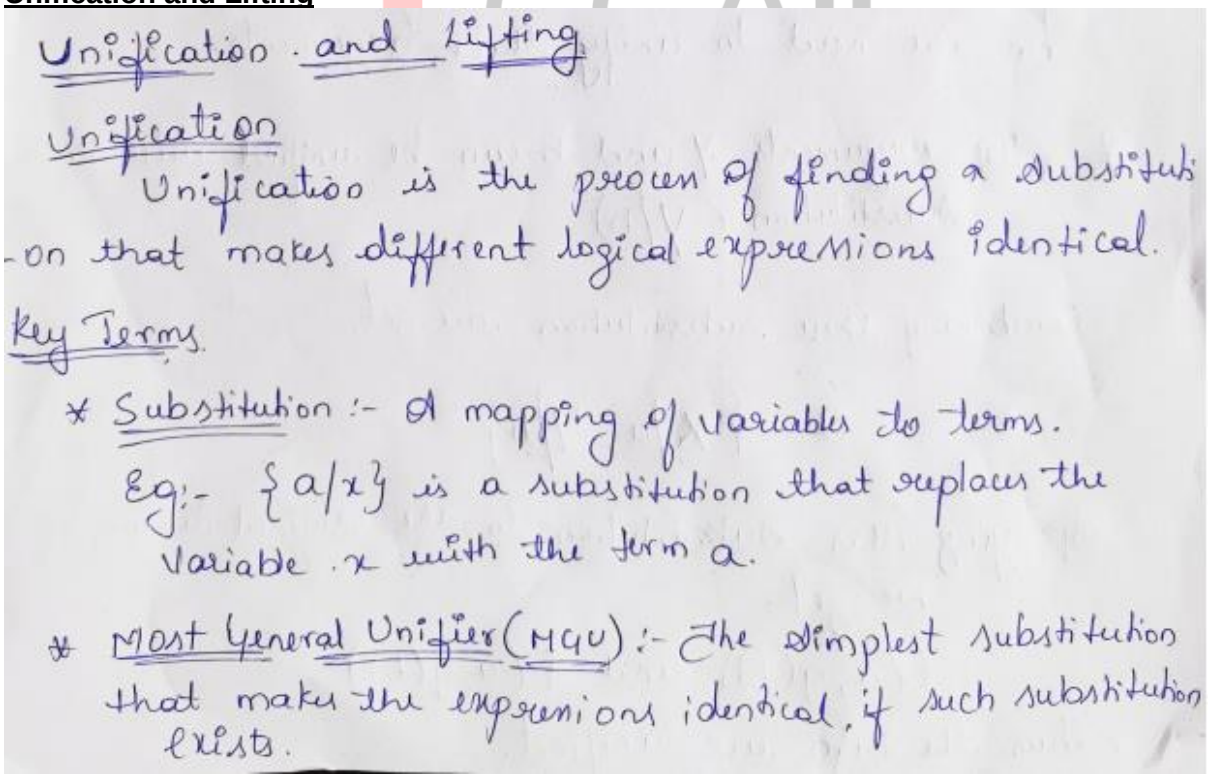
$\forall s \; Breezy(s) \leftrightarrow \exists s' \; Adjacent(s, s') \wedge Pit(s')$

③ Action Effect

- Moving the agent

$\forall s1, s2, t \; At(Agent, s1, t) \wedge Action(Move(s1, s2), t) \rightarrow At(Agent, s2, t+1)$

Unification and Lifting



Unification and Lifting

Unification

Unification is the process of finding a substitution that makes different logical expressions identical.

Key Terms

* Substitution :- A mapping of variables to terms.
  Eg:- $\{a/x\}$ is a substitution that replaces the variable x with the term a.
* Most General Unifier (MGU) :- The simplest substitution that makes the expressions identical, if such substitution exists.

Example

Consider the following terms

1. $f(X, g(Y))$
2. $f(a, g(b))$

To unify these terms, we need to find substitution that makes them identical

* The outer function symbols are both f, so we compare their arguments.
* The first argument X and a can be unified with the substitution (X/a)
* The second arguments $g(Y)$ and $g(b)$ are functions with the same outer function symbol g. So we need to unify their arguments.
* The arguments Y and b can be unified with the substitution (Y/b)

Combining these substitution we get.

$$\theta = \{X/a, Y/b\}$$

Applying this substitution to the original terms, we get.

$f(a, g(b))$ and $f(a, g(b))$

Thus the term are unified.

Lifting

Lifting is the process of converting ground (variable-free) inference rules to work with variables.

This allows us to apply inference rulers more generally, without needing to instantiate all possible ground terms.

Example (1)

Let us the consider Modus Ponens rule:

Ground Version

$$\frac{P \quad P \rightarrow Q}{\therefore Q}$$

Lifted Version

$$\frac{P(x) \quad \forall x(P(x) \rightarrow Q(x))}{\therefore Q(x)}$$

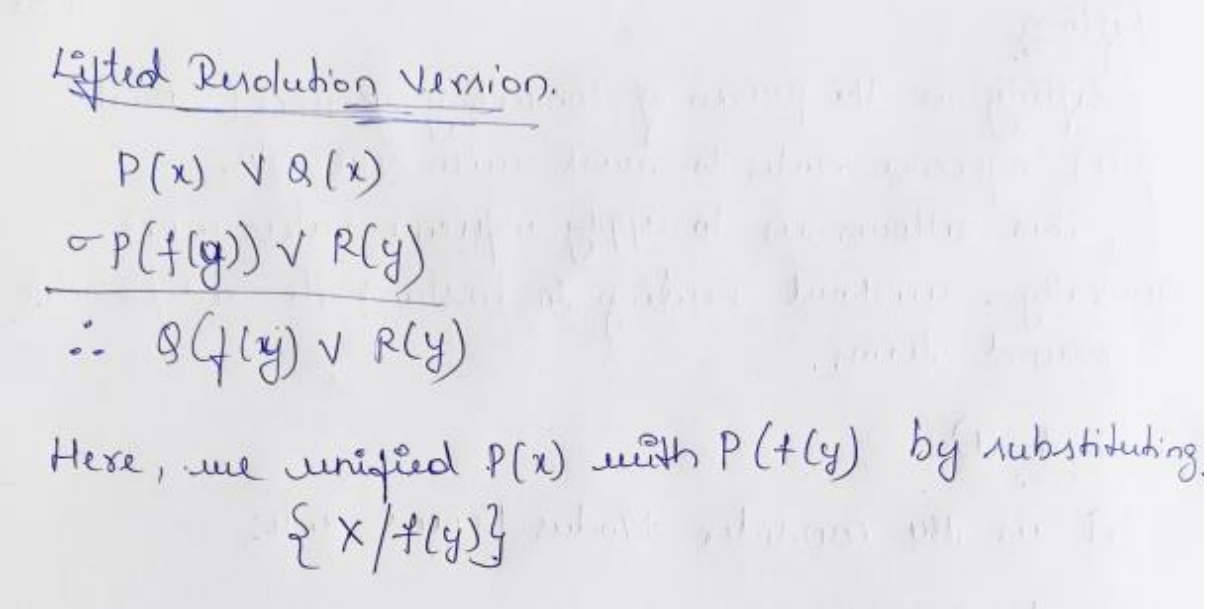
In the lifted version, we can apply the rule to any term x, not just ground terms.

Example (2)

Let us consider the resolution rule.

Ground Version.

$$\frac{A \vee B \quad \neg A \vee C}{\therefore B \vee C.}$$



Inference Engine

- The inference engine is the component of the intelligent system in artificial intelligence, which applies logical rules to the knowledge base to infer new information from known facts.
- The first inference engine was part of the expert system. Inference engine commonly proceeds in two modes, which are:
  - Forward chaining
  - Backward chaining

Horn Clause and Definite clause:

- Horn clause and definite clause are the forms of sentences, which enables knowledge base to use a more restricted and efficient inference algorithm. Logical inference algorithms use forward and backward chaining approaches, which require KB in the form of the first-order definite clause

Definite clause: A clause which is a disjunction of literals with exactly one positive literal is known as a definite clause or strict horn clause.
Horn clause: A clause which is a disjunction of literals with at most one positive literal is known as horn clause. Hence all the definite clauses are horn clauses.
Example: (¬ p V ¬ q V k). It has only one positive literal k.
It is equivalent to p ∧ q → k.

Forward Chaining

Interface College of Computer Applications (ICCA)

Forward Chaining

* Forward chaining is also known as a forward deduction or forward reasoning method when using an inference engine.

* It involves starting with the available data and using inference rules to extract more data until a goal reached

* This method is data-driven, meaning it operates by iteratively applying rules to known facts to produce new facts.

Example

Given Facts

① It is crime for an American to sell weapons to the enemy of America.

② Country Nono is an enemy of America.

③ Nono has some Missiles

④ All the missiles were sold to Nono by colonel

⑤ Missile is a weapon.

⑥ Colonel is American.

Step 1 :- We have to prove that Colonel is a criminal.

Step 2 :- Facts to FOL

① $American(p) \wedge Weapon(y) \wedge Sell(p, y, z) \wedge enemy(z, Ameri$
$\Longrightarrow Criminal(p)$

② $Enemy(Nono, America)$

③ Owns(NoNo, x)
Missile(x)

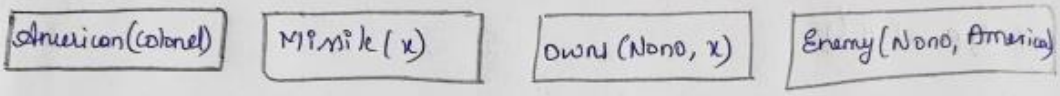
④ ∀x Missile(x) ∧ own(Nono, x) ⇒ sell(colonel, x, Nono)

⑤ Missile(x) ⟹ Weapon(x)

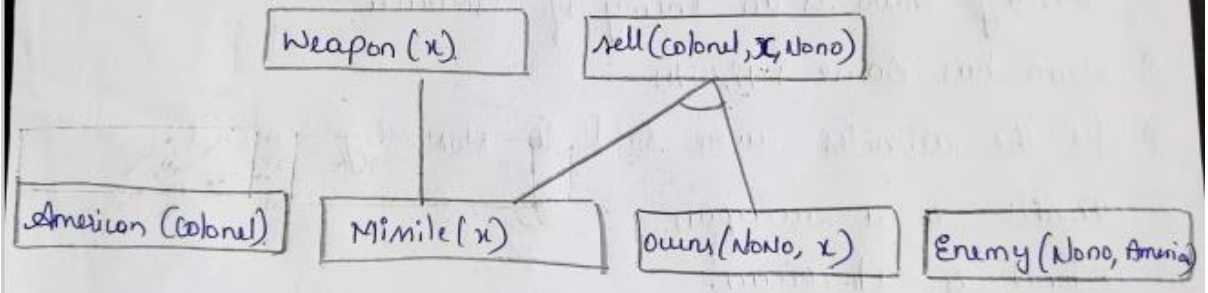
⑥ American(colonel)

Forward chaining proof.

Step 1:- Start with the known facts and will check the sentences which do not have implications.



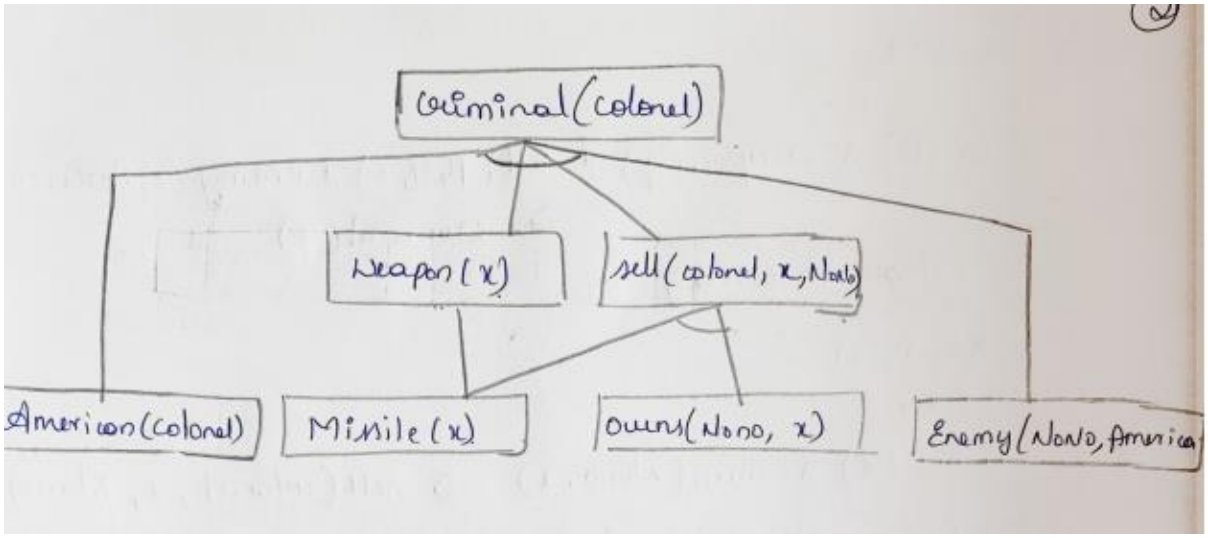
Step 2:- Select those facts which infer from available facts and with satisfied premises.



Step 3:- Rule 1 is satisfied with the substitution.

American(p) → American(colonel).

Weapon(y) → Weapon(x).



Backward Chaining

Backward Chaining

* Backward chaining is also known as a backward deduction or backward reasoning method when using an inference engine.
* A backward chaining algorithm is a form of reasoning which starts with the goal and works backward, chaining through rules to find known facts that support the goal.

Example

Given Facts

1. It is crime for an American to sell weapons to the enemy of America.
2. Country Nono is an enemy of America.
3. Nono has some Missiles.
4. All the Missiles were sold to Nono by colonel.
5. Missile is a weapon.
6. colonel is American.

Proof :- We have to prove that colonel is a criminal.

Facts to FOL:

① American(P) ∧ weapon(y) ∧ sell(P, y, z) ∧ enemy(z, America) ⟹ criminal(P)

② Enemy (Nono, America)

③ Owns (Nono, x)
Missile (x)

④ ∀x Missile(x) ∧ own(Nono, x) ⟹ sell(colonel, x, Nono)

⑤ Missile(x) ⟹ weapon(x)

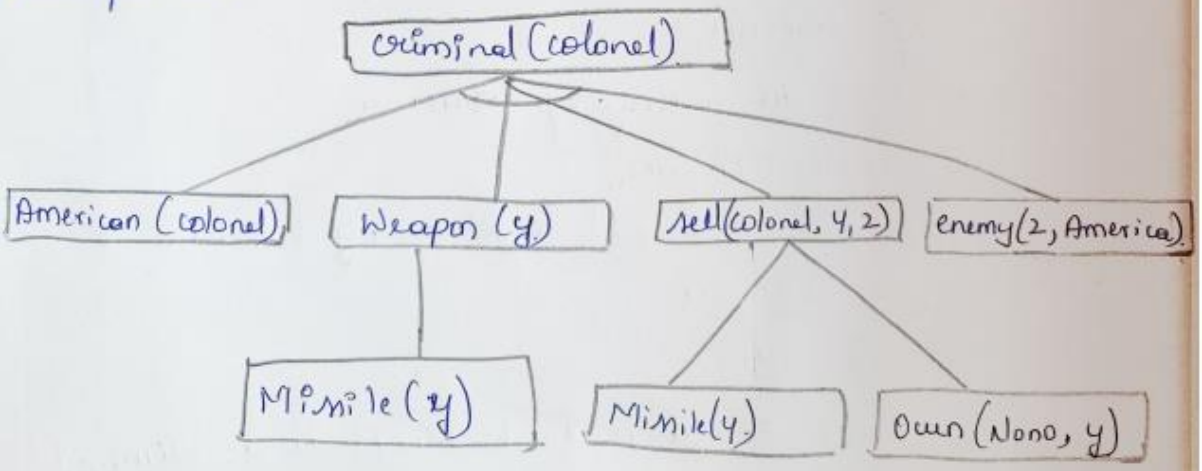
⑥ American (colonel)

Backward Chaining proof:

Step 1:- Consider the goal fact. And from the goal fact, we will infer other facts, we will prove those facts true

criminal (colonel)

Step 2:- We will infer other facts from goal fact which satisfies the rules.



Unit IV: - Learning

Forms of Learning

- Machine learning is a branch of artificial intelligence (AI) and computer science which focuses on the use of data and algorithms to imitate the way that humans learn.

1. Supervised Machine Learning

- Supervised learning is the types of machine learning in which machines are trained using well "labelled" training data, and on basis of that data, machines predict the output. The labelled data means some input data is already tagged with the correct output
- In supervised learning, the training data provided to the machines work as the supervisor that teaches the machines to predict the output correctly.
- It applies the same concept as a student learns in the supervision of the teacher.

Types of supervised Machine learning Algorithms:

- Supervised learning can be further divided into two types of problems:
  - Regression
  - Classification

1. Regression
   - Regression algorithms are used if there is a relationship between the input variable and the output variable.
   - It is used for the prediction of continuous variables, such as Weather forecasting, Market Trends, etc.
   - Some popular Regression algorithms which come under supervised learning: Linear Regression, Regression Trees, Non-Linear Regression, Bayesian Linear Regression, Polynomial Regression
2. Classification
   - Classification algorithms are used when the output variable is categorical, which means there are two classes such as Yes-No, Male-Female, True-false, etc.
   - Some popular Classification algorithms which come under supervised learning Spam Filtering, Random Forest, Decision Trees, Logistic Regression, Support vector Machines.

2. Unsupervised Learning

- Unsupervised learning is a machine learning technique in which models are not supervised using training dataset.
- Instead, models itself find the hidden patterns and insights from the given data.
- It can be compared to learning which takes place in the human brain while learning new things.
- It can be defined as: “Unsupervised learning is a type of machine learning in which models are trained using unlabelled dataset and are allowed to act on that data without any supervision”.

Types of Unsupervised Learning Algorithm:

- The unsupervised learning algorithm can be further categorized into two types of problems:
  - Clustering
  - Association

1. Clustering:
   - Clustering is a method of grouping the objects into clusters such that objects with most similarities remains into a group and has less or no similarities with the objects of another group.
2. Association:

Interface College of Computer Applications (ICCA)

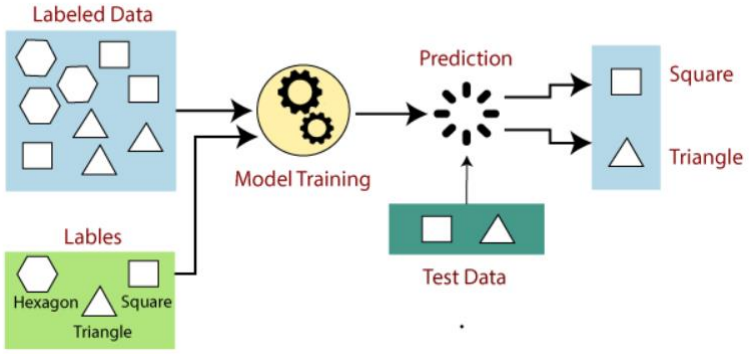
- An association rule is an unsupervised learning method which is used for finding the relationships between variables in the large database.

3.Reinforcement Machine Learning

- Reinforcement learning involves training an agent to make a sequence of decisions by rewarding or punishing it based on its actions. The agent learns to maximize the cumulative reward.
- Reinforcement machine learning algorithm is a learning method that interacts with the environment by producing actions and discovering errors. Trial, error, and delay are the most relevant characteristics of reinforcement learning.
- In this technique, the model keeps on increasing its performance using Reward Feedback to learn the behavior or pattern.
- Some popular Reinforcement algorithms are: Q-Learning, SARSA(State-action-reward-state-action), Deep Q-learning..

Supervised Machine Learning

- Supervised learning is the types of machine learning in which machines are trained using well "labelled" training data, and on basis of that data, machines predict the output. The labelled data means some input data is already tagged with the correct output
- In supervised learning, the training data provided to the machines work as the supervisor that teaches the machines to predict the output correctly.
- It applies the same concept as a student learns in the supervision of the teacher.



Interface College of Computer Applications (ICCA)

- In supervised learning, models are trained using labeled datasets, where the model learns about each type of data.
- Once the training process is completed, the model is tested on the basis of test data (a subset of the training set), and then it predicts the output.

Types of supervised Machine learning Algorithms:

- Supervised learning can be further divided into two types of problems:
  - Regression
  - Classification

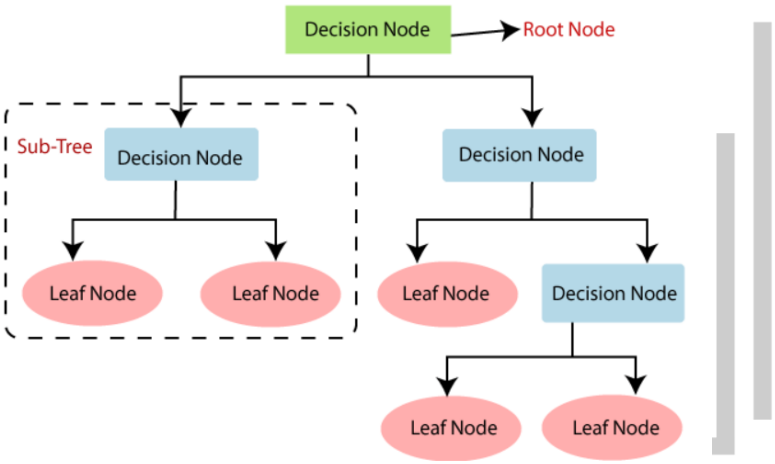
1. Regression

- Regression algorithms are used if there is a relationship between the input variable and the output variable.
- It is used for the prediction of continuous variables, such as Weather forecasting, Market Trends, etc.
- Some popular Regression algorithms which come under supervised learning: Linear Regression, Regression Trees, Non-Linear Regression, Bayesian Linear Regression, Polynomial Regression

2. Classification
   - Classification algorithms are used when the output variable is categorical, which means there are two classes such as Yes-No, Male-Female, True-false, etc.
   - Some popular Classification algorithms which come under supervised learning Spam Filtering, Random Forest, Decision Trees, Logistic Regression, Support vector Machines.

Machine Learning – Decision Tree

- A decision tree is a supervised machine learning algorithm that creates a series of sequential decisions to reach a specific result.
- It is a graphical representation for getting all the possible solutions to a problem/decision based on given conditions.
- It is called a decision tree because, similar to a tree, it starts with the root node, which expands on further branches and constructs a tree-like structure.
- In order to build a tree, we use the CART algorithm, which stands for Classification and Regression Tree algorithm.
- A decision tree simply asks a question, and based on the answer (Yes/No), it further split the tree into subtrees.
- Below diagram explains the general structure of a decision tree:



Decision Tree Terminologies

- Root Node: Represents the entire dataset, which is then split into two or more homogeneous sets.
- Internal Nodes: Represent the features (attributes) of the dataset that are used for splitting the data.
- Leaf Nodes (Terminal Nodes): Represent the final output or decision (class labels in classification or continuous values in regression).
- Branches: Connect nodes, representing the outcome of a test on an attribute.

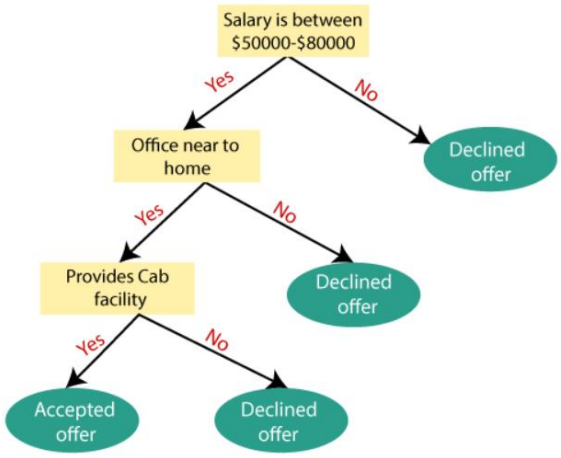
Interface College of Computer Applications (ICCA)

Decision Tree Algorithm Working

Step-1: Begin the tree with the root node, says S, which contains the complete dataset.
Step-2: Find the best attribute in the dataset using Attribute Selection Measure (ASM).
Step-3: Divide the S into subsets that contains possible values for the best attributes.
Step-4: Generate the decision tree node, which contains the best attribute.
Step-5: Recursively make new decision trees using the subsets of the dataset created in step3. Continue this process until a stage is reached where you cannot further classify the nodes and called the final node as a leaf node.

Example:

- Suppose there is a candidate who has a job offer and wants to decide whether he should accept the offer or Not.
- So, to solve this problem, the decision tree starts with the root node (Salary attribute by ASM).
- The root node splits further into the next decision node (distance from the office) and one leaf node based on the corresponding labels.
- The next decision node further gets split into one decision node (Cab facility) and one leaf node.
- Finally, the decision node splits into two leaf nodes (Accepted offers and Declined offer). Consider the below diagram:



Regression and Classification with Linear Model

A linear model is a type of statistical model that assumes a linear relationship between input features (independent variables) and the target variable (dependent variable). Linear models are widely used in both regression and classification tasks.

Linear Regression

- Linear regression is a type of supervised machine learning algorithm that computes the linear relationship between the dependent variable and one or more independent features by fitting a linear equation to observed data.
- When there is only one independent feature, it is known as Simple Linear Regression, and when there are more than one feature, it is known as Multiple Linear Regression.
- When there is only one dependent variable, it is considered Univariate Linear Regression, while when there are more than one dependent variables, it is known as Multivariate Regression.

Interface College of Computer Applications (ICCA)

Types of Linear Regression

1. Simple Linear Regression:
   - In simple linear regression, we model the relationship between a single independent variable X and a dependent variable Y.
   - The goal is to find a linear equation that best predicts Y based on X.
   - The model for simple linear regression is: $y=w_0+w_1x+\epsilon$
     Where:
     y is the dependent variable.
     x is the independent variable.
     w_0 is the intercept (constant term).
     w_1 is the coefficient (slope of the line).
     ϵ is the error term (residual).

Example plot

2. Multiple Linear Regression:
    - Multiple linear regression is an extension of simple linear regression where we have more than one independent variable (feature) to predict the dependent variable (target).
    - It allows us to model the relationship between multiple features and the target variable simultaneously.
    - The model for multiple linear regression is: $\mathbf{y=w_0+w_1x_1+w_2x_2+\cdots+w_nx_n+\epsilon}$
      Where:
      y is the dependent variable.
      $x_1,x_2,\ldots,x_n$ are the independent variables.
      w_0 is the intercept (constant term).
      $w_1,w_2,\ldots,w_n$are the coefficients (slopes for each independent variable).
      ϵ is the error term (residual).

Linear Classification

- Classification with a linear model involves using a linear function to separate data points into different classes.
- Some of the linear classification models are as follows:
    - Logistic Regression
    - Support Vector Machines having kernel = 'linear'
    - Single-layer Perceptron
    - Stochastic Gradient Descent (SGD) Classifier
- One of the most common linear models for classification is logistic regression.
- Linear Decision Boundary**:** In classification, a linear model aims to find a linear decision boundary that separates the classes.
- For two-dimensional data, this is a straight line; for higher-dimensional data, it is a hyperplane.

Interface College of Computer Applications (ICCA)

Logistic Regression**:**

- Unlike linear regression, which predicts continuous values, logistic regression predicts probabilities that a given input belongs to a certain class. The probabilities are then used to make binary decisions (e.g., class 0 or class 1).

Equation of Logistic Regression:

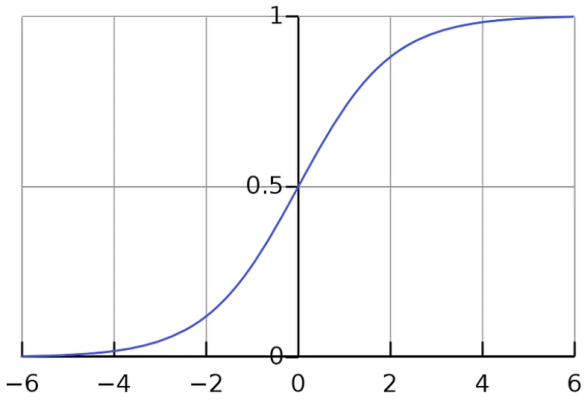
- When it comes to classification, we are determining the probability of an observation to be part of a certain class or not.
- Therefore, we wish to express the probability with a value between 0 and 1.

- A probability close to 1 means the observation is very likely to be part of that category.
- In order to generate values between 0 and 1, we express the probability using this equation:

$$p(X) = \frac{\exp(\beta_0 + \beta_1 X)}{1 + \exp(\beta_0 + \beta_1 X)}$$

Sigmoid Function

- The equation above is defined as the sigmoid function.
- The above equation when plotted will always results in a S-shaped curve bound between 0 and 1.



Logistic Regression Curve

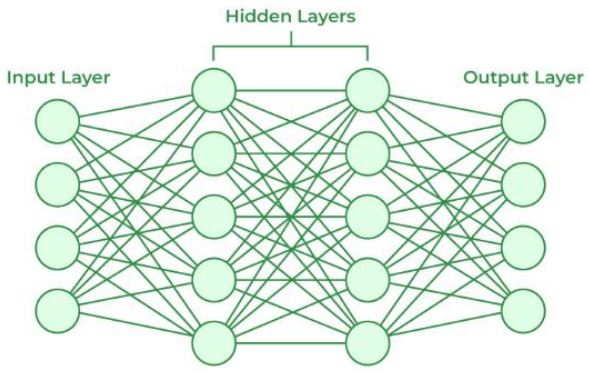
Artificial Neural Networks

- An Artificial Neural Network (ANN) is a computational model inspired by the way biological neural networks in the human brain process information.
- ANNs are used in a variety of machine learning tasks, including classification, regression, and pattern recognition.
- In simple words, Neural Networks are a set of algorithms that tries to recognize the patterns, relationships, and information from the data through the process which is inspired by and works like the human brain/biology.

Interface College of Computer Applications (ICCA)

Components / Architecture of Neural Network

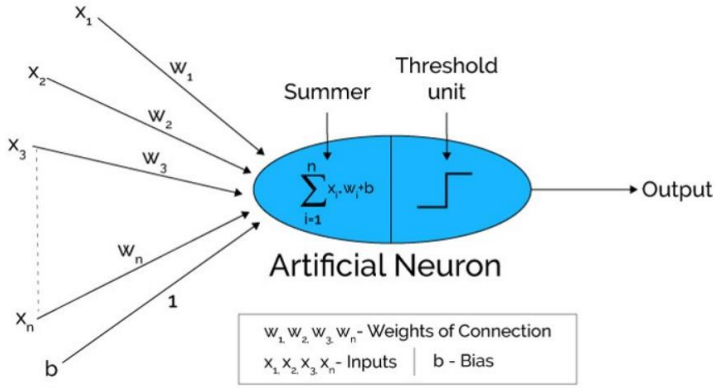
- A simple neural network consists of three components :
  - Input layer
  - Hidden layer
  - Output layer



Neural Networks Architecture

1. Input Layer: Also known as Input nodes are the inputs/information from the outside world is provided to the model to learn and derive conclusions from. Input nodes pass the information to the next layer i.e Hidden layer.
2. Hidden Layer: Hidden layer is the set of neurons where all the computations are performed on the input data. There can be any number of hidden layers in a neural network. The simplest network consists of a single hidden layer.
3. Output layer: The output layer is the output/conclusions of the model derived from all the computations performed. There can be single or multiple nodes in the output layer. If we have a binary classification problem the output node is 1 but in the case of multi-class classification, the output nodes can be more than 1.

Step by Step Working of the Artificial Neural Network



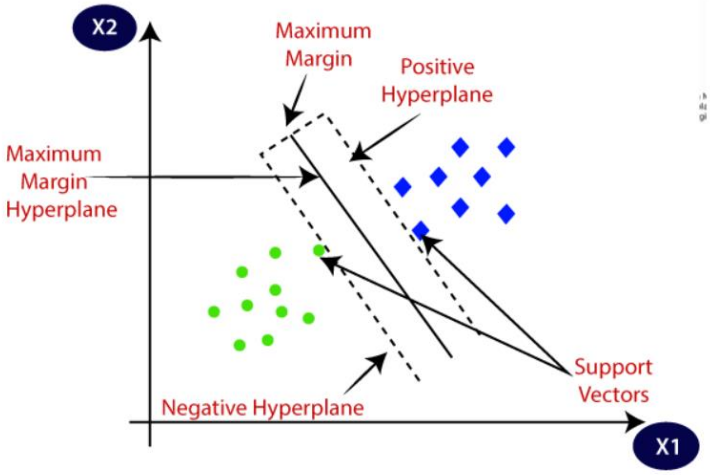
1. In the first step, Input units are passed i.e data is passed with some weights attached to it to the hidden layer. In the above image inputs x1,x2,x3,….xn is passed.
2. Each hidden layer consists of neurons. All the inputs are connected to each neuron.
3. After passing on the inputs, all the computation is performed in the hidden layer
4. Computation performed in hidden layers are done in two steps which are as follows :
    a. First of all, all the inputs are multiplied by their weights. Weight is the gradient or coefficient of each variable. It shows the strength of the particular input. After assigning the weights, a bias variable is added. Bias is a constant that helps the model to fit in the best way possible.
    Z1 = W1*In1 + W2*In2 + W3*In3 + W4*In4 + W5*In5 + b
    Where, W1, W2, W3, W4, W5 are the weights assigned to the inputs In1, In2, In3, In4, In5, and b is the bias.
    b. Then in the second step, the activation function is applied to the linear equation Z1. The activation function is a nonlinear transformation that is applied to the input before sending it to the next layer of neurons. The importance of the activation function is to inculcate nonlinearity in the model. There are several activation functions that will be listed in the next section.
5. After passing through every hidden layer, we move to the last layer i.e. our output layer which gives us the final output.
    The process explained above is known as forwarding Propagation.
6. After getting the predictions from the output layer, the error is calculated i.e. the difference between the actual and the predicted output.
7. If the error is large, then the steps are taken to minimize the error and for the same purpose, Back Propagation is performed.

Note: - Back Propagation is the process of updating and finding the optimal values of weights or coefficients which helps the model to minimize the error i.e difference between the actual and predicted values.

Interface College of Computer Applications (ICCA)

Support Vector Machine

- Support Vector Machine or SVM is one of the most popular Supervised Learning algorithms, which is used for Classification as well as Regression problems. However, primarily, it is used for Classification problems in Machine Learning.
- The goal of the SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best decision boundary is called a hyperplane.



Important Components

1. Hyperplane:
   - A hyperplane is a decision boundary that separates different classes in the feature space.
   - In a 2-dimensional space, this hyperplane is a line, and in 3-dimensional space, it's a plane. For higher dimensions, it's called a hyperplane.
2. Support Vectors:
   - Support vectors are the data points that are closest to the hyperplane. These points are critical in defining the position and orientation of the hyperplane.
   - They lie on the edge of the margin and are instrumental in maximizing the margin.
3. Margin:
   - The margin is the distance between the hyperplane and the nearest data points from either class.
   - SVM aims to maximize this margin. This concept is known as Maximum Margin Classification.

ICCA
Interface College of Computer Applications (ICCA)

Types of SVM:

1. Linear SVM:
   - Used when the data is linearly separable, meaning it can be separated by a straight line (or hyperplane in higher dimensions).
2. Non-Linear SVM:
   - Used when the data is not linearly separable. In this case, SVM uses a technique called the kernel trick to transform the data into a higher dimension where a hyperplane can be used to separate the classes.

Steps to Implement SVM

1. Data Preparation:
   - Collect and clean your dataset.

- Split the data into features (X) and target labels (y).
- Normalize or standardize the features (important for SVMs).
- Split the data into training and testing sets.

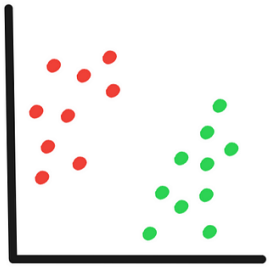
2. Kernel Selection:
   - Choose an appropriate kernel function based on your data:
   - Linear kernel for linearly separable data.
   - Polynomial kernel for more complex, non-linear data.
   - Radial Basis Function (RBF) kernel for highly non-linear data.
   - Custom kernels for specific problem domains
3. Set the Parameters:
   - Regularization parameter (C) controls the trade-off between achieving a low error on the training data and minimizing the norm of the weights.
   - Kernel-specific parameters like gamma in RBF kernel.
4. Train the Model:
   - Use the training data to find the optimal hyperplane that maximizes the margin.
5. Make Predictions:
   - Use the hyperplane to classify new data points.

Types SVM

3. Linear SVM:
   - Used when the data is linearly separable, meaning it can be separated by a straight line (or hyperplane in higher dimensions).

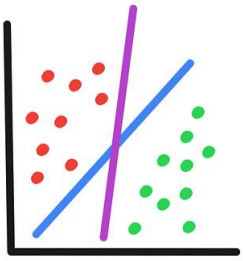
Working of Linear SVM

Consider a dataset with two set of balls, red and green. The dataset has two features, p1 and p2. We want to classify the coordinate pair (p1,p2) into either red balls or green balls. The image is shown below:



Interfac er Applications (ICCA)

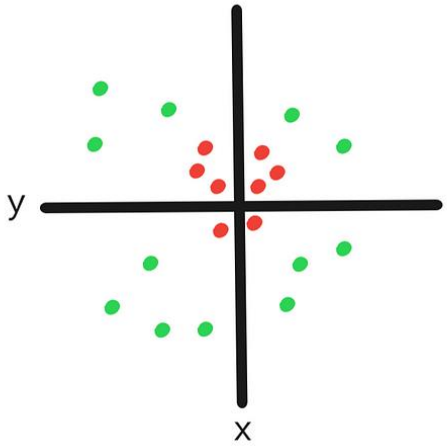
As it is a 2-dimensional space, we can easily separate these two classes by using a straight line. But there can be multiple lines that can separate these classes. Consider the below image:



Hence, the SVM algorithm helps to find the best line or decision boundary, called the hyperplane. SVM finds the closest point of the lines from both the classes. These points are called support vectors.

4. Non-Linear SVM:
   - Used when the data is not linearly separable. In this case, SVM uses a technique called the kernel trick to transform the data into a higher dimension where a hyperplane can be used to separate the classes.

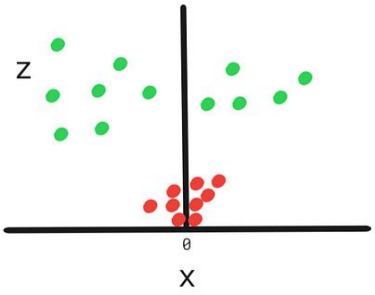
If data is linearly arranged, then we can separate it by using linear SVM, but for non-linear data, we cannot draw a single straight line.
Consider the below image:



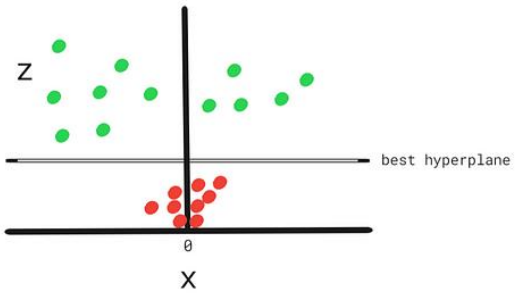
To separate these data points, we need to add one more dimension. For linear data, we used two dimensions x and y, so for non-linear data, we will add a third dimension z.
It can be calculated as:

z = x^2 + y^2

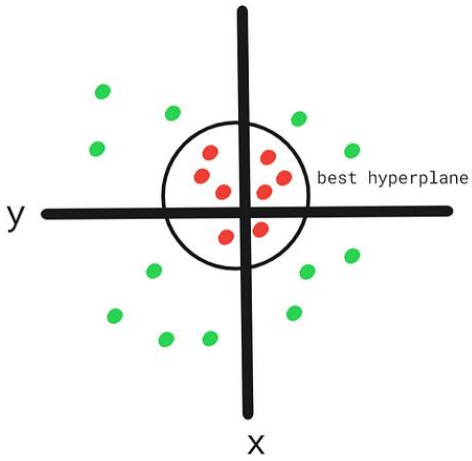
By adding the third dimension, the sample space will become as below image:



Now we will use SVM to divide the datasets into two classes in the following way.



Since we are in 3-dimensional space, it looks like a plane parallel to the x-axis. If we convert it in 2-dimensional space with z=1, then it can be depicted as:



Interface College of Computer Applications (ICCA)

Unit V: - Applications of AI

Applications of AI

- Healthcare: AI is used for medical image analysis, drug discovery, personalized treatment plans, and patient monitoring. It helps in early disease detection, virtual health assistants, and predicting patient outcomes.
- Finance: In finance, AI is used for fraud detection, algorithmic trading, credit scoring, risk assessment, and customer service automation. It enables financial institutions to make data-driven decisions and improve operational efficiency.
- Retail: AI is applied in retail for personalized recommendations, demand forecasting, inventory management, supply chain optimization, and customer service through chatbots. It enhances the shopping experience and helps businesses understand consumer behavior.
- Manufacturing: AI is used in manufacturing for predictive maintenance, quality control, supply chain management, and process optimization. It enables automation of repetitive tasks, improves productivity, and reduces downtime.
- Transportation: AI plays a crucial role in autonomous vehicles, traffic management systems, route optimization, predictive maintenance for vehicles and infrastructure, and logistics optimization. It enhances safety, efficiency, and sustainability in transportation.
- Education: AI is used in education for personalized learning, adaptive tutoring systems, automated grading, and student engagement analysis. It helps in creating tailored learning experiences and improving educational outcomes.
- Marketing: AI is applied in marketing for customer segmentation, sentiment analysis, content optimization, and targeted advertising. It enables marketers to reach the right audience with the right message at the right time.
- Natural Language Processing (NLP): NLP techniques are used in various applications such as language translation, sentiment analysis, chatbots, virtual assistants, and text summarization. It enables computers to understand and generate human language.
- Computer Vision: Computer vision is used for object detection, image classification, facial recognition, autonomous vehicles, medical image analysis, and surveillance. It enables machines to interpret and understand visual information.
- Entertainment: AI is used in the entertainment industry for content recommendation, personalized streaming services, content creation (such as AI-generated music or art), and gaming (including AI-driven NPCs and procedural content generation).

Interface College of Computer Applications (ICCA)

Natural Language Processing(NLP)

- Natural Language Processing (NLP) is a subfield of artificial intelligence (AI) that focuses on the interaction between computers and humans through natural language.
- It encompasses the development of algorithms and techniques to enable computers to understand, interpret, and generate human language in a meaningful way.

Applications of NLP

1. Text and Document Processing:
   - Text Classification: Categorize texts into predefined categories (e.g., spam detection).
   - Document Summarization: Automatically generate concise summaries of longer texts.
   - Sentiment Analysis: Determine the sentiment expressed in text (positive, negative, neutral).

2. Information Retrieval:

- Search Engines: Improve search results by understanding query context and intent.
- Question Answering Systems: Provide accurate answers to user queries based on text analysis.
- Document Retrieval: Find relevant documents based on keywords or queries.

3. Machine Translation:
   - Language Translation: Translate text or speech from one language to another.
   - Localization: Adapt content to different languages and cultural contexts for global audiences.
4. Speech Recognition and Synthesis:
   - Voice Assistants: Convert spoken language into text and vice versa (e.g., Siri, Alexa, Google Assistant).
   - Transcription Services: Convert speech into written text.
   - Accessibility: Provide text-to-speech and speech-to-text services for accessibility.
5. Named Entity Recognition (NER):
   - Identify and classify named entities in text (e.g., names of people, organizations, locations).
   - Used in information extraction tasks, such as news categorization or data mining.
6. Text Generation:
   - Generate text automatically based on given prompts or contexts.
   - Applications include chatbots, automated content creation, and creative writing assistance.
7. Information Extraction:
   - Extract structured information from unstructured text (e.g., extracting dates, locations, events from news articles).
   - Used in data mining, business intelligence, and knowledge graph construction.
8. Natural Language Understanding (NLU):
   - Analyze and understand the meaning and context of human language.
   - Enables systems to interpret user intents and respond appropriately (e.g., in dialogue systems).
9. Semantic Analysis:
   - Analyze the meaning of words, phrases, and sentences to understand relationships between them.
   - Important for tasks like semantic search, automated question answering, and language modeling.
10. Social Media Analysis:
    - Analyze and understand trends, sentiments, and user interactions on social media platforms.
    - Applications include brand monitoring, sentiment analysis of customer feedback, and targeted advertising.

Interface College of Computer Applications (ICCA)

Text Classification

- Text classification is a natural language processing (NLP) task that involves categorizing text documents into predefined categories or classes based on their content. It is a fundamental problem in NLP and has numerous applications across various domains.

General steps in Text Classification

1. Problem Definition:
   - Define the classification task clearly (e.g., sentiment analysis, topic categorization).
   - Identify the categories or labels you want to assign.
   - Determine the type of classification (binary, multi-class, multi-label).
2. Data Collection:

- Gather a dataset of text documents along with their corresponding labels or categories. This dataset should be representative of the task you want to perform.

3. Data Preprocessing:
   - Clean and preprocess the text data to prepare it for analysis. This may involve steps such as:
   (1) Text cleaning:
      (a) Remove HTML tags, if any.
      (b) Handle special characters and numbers.
      (c) Convert text to lowercase.
   (2) Tokenization: Split text into smaller units (words, subwords, or characters).
   (3) Stopword removal: Eliminate common words that don't carry much meaning.
   (4) Stemming or lemmatization: Reduce words to their root form(running to run).
   (5) Handling missing data or noise.
4. Feature Extraction / Vectorization:
   - Convert text data into numerical features that machine learning algorithms can process.
   - Common approaches include: Bag-of-Words (BoW), Term Frequency-Inverse Document Frequency (TF-IDF), Document embeddings (Doc2Vec) etc.
5. Splitting the Dataset:
   - Divide the data into training, validation, and test sets.
6. Model Selection:
   - Choose a suitable algorithm based on the problem, dataset size, and complexity.
   - Options range from traditional machine learning (Naive Bayes, SVM, Logistic Regression) to deep learning (RNNs, CNNs, Transformers).
7. Model Training:
   - Feed the training data into the chosen model.
   - The model learns to associate input features with output labels.
8. Model Evaluation:
   - Assess the model's performance on the validation set.
   - Use metrics such as accuracy, precision, recall, F1-score, and confusion matrix.
   - For multi-class problems, consider macro, micro, or weighted averages of these metrics.
9. Hyperparameter Tuning:
   - Adjust model hyperparameters to improve performance.
   - This might involve grid search, random search, or Bayesian optimization.
   - Re-evaluate the model after each tuning iteration.
10. Testing:
   - Once satisfied with the model's performance on the validation set, evaluate it on the test set.
   - This gives an estimate of how well the model will perform on unseen data.
11. Deployment:
   - Integrate the trained model into the target application or system.
   - This might involve creating an API, a web service, or embedding the model directly into an application.

Application of Text Classification

- Text classification has numerous applications across various domains. Here are some common applications:

1. Spam Filtering: Classifying emails as spam or non-spam to protect users from unwanted messages. Text classification algorithms can analyze the content and metadata of emails to determine whether they are likely to be spam.

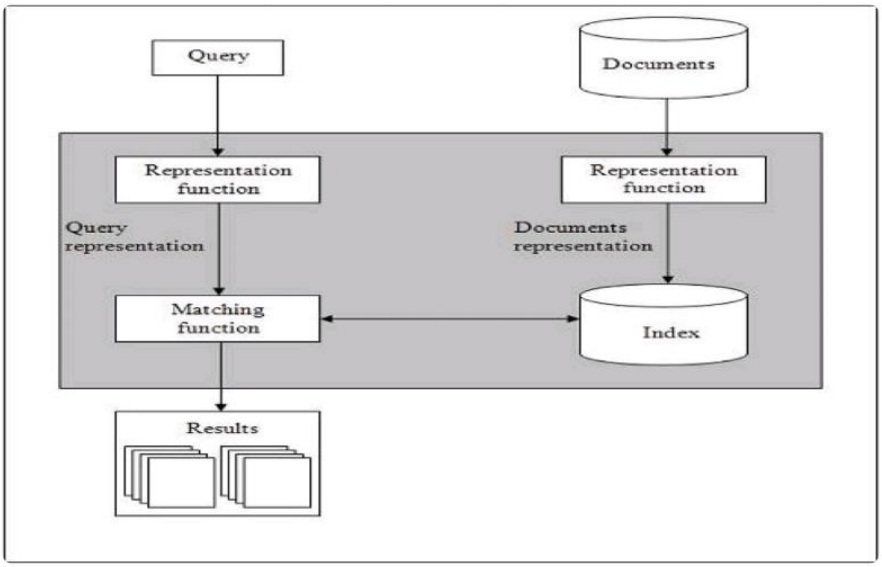
Interface College of Computer Applications (ICCA)

2. Sentiment Analysis: Identifying the sentiment or opinion expressed in text data, such as product reviews, social media posts, or customer feedback. Sentiment analysis can help businesses understand customer sentiment towards their products or services and make data-driven decisions.
3. Topic Categorization: Categorizing text documents into predefined topics or categories, such as news articles, blog posts, or research papers. Topic categorization enables efficient organization and retrieval of information in large text corpora.
4. Language Detection: Identifying the language of a given text document. Language detection is useful for multilingual applications, such as language-specific content filtering or language-based user segmentation.
5. Intent Recognition: Understanding the intent behind user queries or commands in natural language. Intent recognition is used in chatbots, virtual assistants, and voice-controlled devices to interpret user input and provide appropriate responses or actions.
6. Document Classification: Classifying documents based on their content, such as legal documents, medical records, or financial reports. Document classification enables automated document organization, retrieval, and management.
7. Fake News Detection: Detecting fake or misleading news articles by classifying them based on their content, source, and other features. Fake news detection algorithms analyze textual features and metadata to identify misinformation and disinformation.
8. Medical Text Analysis: Analyzing medical records, clinical notes, and biomedical literature for tasks such as disease classification, patient risk assessment, and adverse event detection. Medical text analysis helps healthcare professionals make informed decisions and improve patient outcomes.
9. Legal Document Analysis: Analyzing legal texts, contracts, and court opinions for tasks such as document summarization, case law prediction, and legal document classification. Legal document analysis assists legal professionals in legal research and decision-making.
10. Social Media Monitoring: Monitoring social media platforms for trends, sentiment, and emerging topics related to brands, products, or events. Social media monitoring tools use text classification to analyze and categorize social media posts, comments, and conversations.

Information Retrieval

- Information retrieval (IR) may be defined as a software program that deals with the organization, storage, retrieval and evaluation of information from document repositories particularly textual information.
- The system assists users in finding the information they require but it does not explicitly return the answers of the questions.
- It informs the existence and location of documents that might consist of the required information.
- The documents that satisfy user's requirement are called relevant documents.
- A perfect IR system will retrieve only relevant documents.

Interface College of Computer Applications (ICCA)



Steps involved in Information retrieval

1. Query Input:
   - The process starts with a user entering a query. This is the information that the user is seeking.
2. Query Representation:
   - The query is then processed by a "Representation Function" which transforms the query into a suitable format for comparison with the document representations.
3. Document Collection:
   - A set of documents (or a document database) is stored in the system. These are the documents that will be searched to find the ones most relevant to the user's query.
4. Document Representation:
   - Each document in the document collection is processed by a "Representation Function" to convert it into a structured format (often a vector) that can be indexed and compared with the query representation.
5. Indexing:
   - During indexing, key features of each document are extracted. These could be words, phrases, named entities, or other meaningful units. Indexing involves: Tokenization of string, Removing frequent words, Stemming etc.
6. Matching Function:
   - The "Matching Function" compares the query representation to the document representations stored in the index. This comparison uses various algorithms to determine the relevance of each document to the query. Common methods include cosine similarity, tf-idf weighting, or more advanced machine learning-based approaches.
7. Retrieval of Results:
   - Based on the results of the matching function, a set of documents is retrieved. These documents are considered the most relevant to the user's query.
8. Output Results:
   - Finally, the retrieved documents are presented as results to the user. These results are often ranked by relevance so that the most relevant documents appear first.

Applications of Information Retrieval

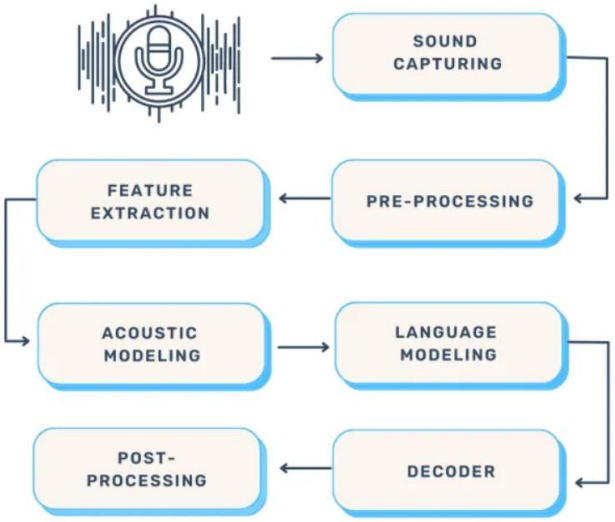
Interface College of Computer Applications (ICCA)

1. Search Engines:
    - Web Search Engines: Google, Bing, and Yahoo use advanced IR techniques to crawl, index, and retrieve web pages relevant to user queries.
    - Enterprise Search: Internal search engines within organizations that help employees find documents, emails, and other internal data.
2. Digital Libraries:
    - IR systems are used to organize and retrieve books, academic papers, and other scholarly articles in digital libraries such as JSTOR, IEEE Xplore, and Google Scholar.
3. E-commerce:
    - Product search and recommendation systems on platforms like Amazon and eBay use IR to help users find products and suggest items based on user behavior and preferences.
4. Multimedia Retrieval:
    - Systems that allow users to search for images, videos, and audio files based on content features like text annotations, metadata, and visual similarities (e.g., Google Images, YouTube).
5. Healthcare:
    - Medical Information Systems: Retrieving patient records, research papers, and clinical guidelines.
6. Social Media:
    - Platforms like Facebook, Twitter, and Instagram use IR to help users find relevant posts, people, and topics. Hashtag and keyword searches are common examples.
7. Legal Information Retrieval:
    - Systems that assist legal professionals in finding case laws, statutes, legal opinions, and other legal documents (e.g., LexisNexis, Westlaw).
8. Recommendation Systems:
    - Used by streaming services like Netflix and Spotify to recommend movies, shows, music, and other media based on user preferences and past behavior.
9. Customer Support:
    - Chatbots and virtual assistants use IR to retrieve relevant answers from a knowledge base to respond to customer queries (e.g., FAQ systems, automated customer service).
10. Personal Assistants:
    - Virtual assistants like Siri, Alexa, and Google Assistant use IR to understand and process user queries, retrieve relevant information, and provide accurate responses.

Interface College of Computer Applications (ICCA)

Speech Recognition

Speech recognition, also known as automatic speech recognition (ASR), speech-to-text (STT), and computer speech recognition, is a technology that enables a computer to recognize and convert spoken language into text.



Features of speech recognition systems

- Speech recognition systems have several components that work together to understand and process human speech.
- Key features of effective speech recognition are:

1. Audio preprocessing: After you have obtained the raw audio signal from an input device, you need to preprocess it to improve the quality of the speech input The main goal of audio preprocessing is to capture relevant speech data by removing any unwanted artifacts and reducing noise.
2. Feature extraction: This stage converts the preprocessed audio signal into a more informative representation. This makes raw audio data more manageable for machine learning models in speech recognition systems.
3. Language model weighting: Language weighting gives more weight to certain words and phrases, such as product references, in audio and voice signals. This makes those keywords more likely to be recognized in a subsequent speech by speech recognition systems.
4. Acoustic modeling: It enables speech recognizers to capture and distinguish phonetic units within a speech signal. Acoustic models are trained on large datasets containing speech samples from a diverse set of speakers with different accents, speaking styles, and backgrounds.
5. Speaker labeling: It enables speech recognition applications to determine the identities of multiple speakers in an audio recording. It assigns unique labels to each speaker in an audio recording, allowing the identification of which speaker was speaking at any given time.
6. Profanity filtering: The process of removing offensive, inappropriate, or explicit words or phrases from audio data.

Interface College of Computer Applications (ICCA)

Applications of Speech Recognition

1. Virtual Assistants
   - Software agents that perform tasks or services based on voice commands.
   - Example: Siri can set reminders, send messages, and answer questions when you speak to it.
2. Transcription Services
   - Converting spoken language into written text.
   - Example: Otter.ai transcribes meetings and interviews into text, making it easy to review and share notes.

3. Customer Service
   - Automated systems that handle customer inquiries using voice recognition.
   - Example: An airline's automated phone system can understand and respond to questions about flight statuses and bookings.
4. Healthcare
   - Using voice recognition to assist medical professionals with documentation.
   - Example: Dragon Medical One allows doctors to dictate patient notes directly into electronic health records.
5. Accessibility
   - Tools that help individuals with disabilities interact with technology through voice commands.
   - Example: Voice control features on smartphones enable users with motor impairments to operate their devices hands-free.
6. Automotive Industry
   - Voice-activated systems in vehicles for hands-free control.
   - Example: A driver can use voice commands to set the GPS destination or make phone calls without taking their hands off the wheel.
7. Education
   - Applications that help with learning and practice using speech recognition.
   - Example: Duolingo uses voice input to help users practice pronunciation in language learning exercises.
8. Entertainment
   - Using voice commands to control media playback and search for content.
   - Example: Saying "Play the latest episode of my favorite show" on a streaming service to start watching immediately.
9. Smart Home Devices
   - Devices that can be controlled using voice commands for convenience and automation.
   - Example: Telling Amazon Echo to turn off the lights or adjust the thermostat.
10. Security
    - Using voice recognition to verify identities.
    - Example: Some banking apps use voice biometrics to authenticate users, ensuring secure access to accounts.
11. Navigation
    - Providing spoken directions and commands in navigation systems.
    - Example: A GPS system offering voice-guided directions, allowing drivers to keep their eyes on the road.

Interface College of Computer Applications (ICCA)

Image Processing

- Digital image processing is the use of algorithms and mathematical models to process and analyze digital images.
- The goal of digital image processing is to enhance the quality of images, extract meaningful information from images, and automate image-based tasks.

The basic steps involved in digital image processing are:

1. Image acquisition: This involves capturing an image using a digital camera or scanner, or importing an existing image into a computer.
2. Image enhancement: This involves improving the visual quality of an image, such as increasing contrast, reducing noise, and removing artifacts.
3. Image restoration: This involves removing degradation from an image, such as blurring, noise, and distortion.

4. Image segmentation: This involves dividing an image into regions or segments, each of which corresponds to a specific object or feature in the image.
5. Image representation and description: This involves representing an image in a way that can be analyzed and manipulated by a computer, and describing the features of an image in a compact and meaningful way.
6. Image analysis: This involves using algorithms and mathematical models to extract information from an image, such as recognizing objects, detecting patterns, and quantifying features.
7. Image synthesis and compression: This involves generating new images or compressing existing images to reduce storage and transmission requirements.

Applications of Image processing

1. Medical Imaging
   - MRI and CT Scans: Enhance images for better diagnosis.
   - X-ray Imaging: Improve image quality and detect fractures or abnormalities.
   - Ultrasound Imaging: Enhance clarity for better interpretation.
   - Examples: Tumor Detection, Bone Fracture Detection, Retinal Image Analysis

2. Surveillance and Security
   - Facial Recognition: Identify individuals in security footage.
   - Motion Detection: Trigger alerts based on movement in restricted areas.
   - License Plate Recognition: Automated systems to read and log vehicle plates.
   - Examples: Access Control, Intrusion Detection, Traffic Management

3. Industrial Automation
   - Quality Control: Inspect products on assembly lines.
   - Robotic Vision: Enable robots to perform tasks with visual input.
   - Barcode and QR Code Scanning: Automate inventory and logistics management.
   - Examples: Defect Detection, Pick-and-Place Robots, Automated Sorting

4. Remote Sensing
   - Satellite Imaging: Monitor environmental changes, urban development, and natural disasters.
   - Aerial Photography: Used in agriculture, forestry, and land management.
   - Weather Forecasting: Analyze cloud patterns and atmospheric conditions.
   - Examples:Agricultural Monitoring, Deforestation Tracking, Disaster Management

Interface College of Computer Applications (ICCA)

5. Automotive Industry
   - Autonomous Driving: Enable self-driving cars to interpret and respond to their environment.
   - Driver Assistance Systems: Lane departure warnings, adaptive cruise control, and parking assistance.
   - Traffic Sign Recognition: Detect and interpret traffic signs.
   - Examples: Self-driving Cars, Advanced Driver-Assistance Systems (ADAS), Traffic Monitoring.

6. Entertainment and Media
   - Image and Video Enhancement: Improve quality of photos and videos.
   - Special Effects: Create realistic effects for movies and video games.
   - Augmented Reality (AR) and Virtual Reality (VR): Integrate digital information with the user's environment.

- Examples: Photo Editing, Film Production, Interactive Games

7. Biometric Systems
   - Fingerprint Recognition: Identify individuals based on fingerprint patterns.
   - Iris Recognition: Use unique patterns in the iris for secure authentication.
   - Voice Recognition: Identify or verify users through voice patterns.
   - Examples: Secure Authentication, Time and Attendance Systems, Personal Identification.

8. Agriculture
   - Precision Farming: Optimize farming practices using detailed image analysis.
   - Crop Monitoring: Assess the health and growth of crops.
   - Pest Detection: Identify and control pest infestations.
   - Examples: Drone Surveillance, Soil Analysis, Yield Prediction

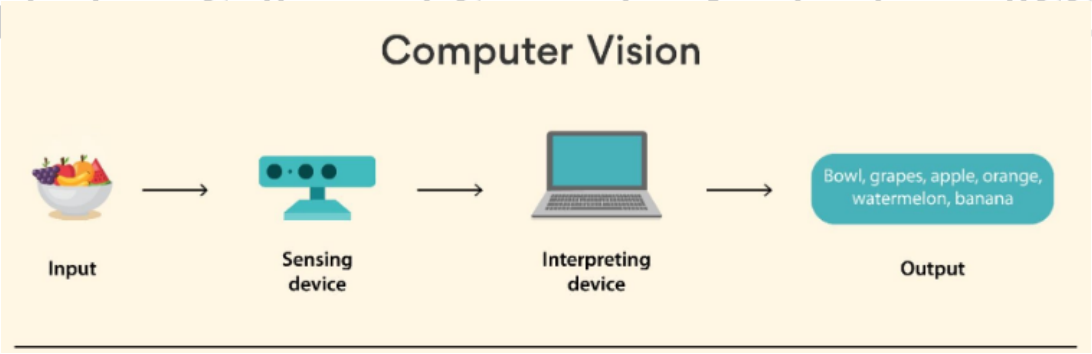
9. Document Processing
   - Optical Character Recognition (OCR): Convert scanned documents into editable text.
   - Form Processing: Automatically extract data from forms and invoices.
   - Handwriting Recognition: Convert handwritten text into digital format.
   - Examples: Automated Data Entry, Archiving, Cheque Processing

10. Scientific Research
    - Microscopy Image Analysis: Enhance and analyze microscopic images.
    - Astronomical Imaging: Process images from telescopes to study celestial objects.
    - Particle Tracking: Analyze motion of particles in physics experiments.
    - Examples: Cell Counting, Galaxy Classification, Flow Analysis

Computer Vision

- Computer vision is a field of study in artificial intelligence that focuses on enabling computers to interpret and understand the visual world in the same way that humans do.
- It involves developing algorithms and models that can analyze digital images and videos to recognize and classify objects, track their movements, and extract useful information from visual data.

Interface College of Computer Applications (ICCA)



Basic steps involved in Computer Vision

1. Image acquisition: This involves capturing digital images or videos using cameras, sensors, or other devices.

2. Preprocessing: The raw data from the images is preprocessed, which involves tasks like noise reduction, image enhancement, and normalization.

3. Feature extraction: The algorithm extracts important features from the image, such as edges, corners, and textures. These features help to identify and classify objects in the image.
4. Object recognition and tracking: The algorithm uses the extracted features to identify and recognize objects in the image, and can also track their movements over time.
Interpretation and decision making: The algorithm analyzes the identified objects and their movements to extract useful information and make decisions or predictions based on that information.

Difference between Image Processing and Computer Vision

Image Processing	Computer Vision
Image processing is mainly focused on processing the raw input images to enhance them or preparing them to do other tasks	Computer vision is focused on extracting information from the input images or videos to have a proper understanding of them to predict the visual input like human brain.
Image processing uses methods like Anisotropic diffusion, Hidden Markov models, Independent component analysis, Different Filtering etc.	Image processing is one of the methods that is used for computer vision along with other Machine learning techniques, CNN etc.
Image Processing is a subset of Computer Vision.	Computer Vision is a superset of Image Processing.
Examples of some Image Processing applications are- Rescaling image (Digital Zoom), Correcting illumination, Changing tones etc.	Examples of some Computer Vision applications are- Object detection, Face detection, Hand writing recognition etc.

Robotics

- Robotics, design, construction, and use of machines (robots) to perform tasks done traditionally by human beings.
- Robots are widely used in such industries as automobile manufacture to perform simple repetitive tasks, and in industries where work must be performed in environments hazardous to humans.
- Many aspects of robotics involve artificial intelligence; robots may be equipped with the equivalent of human senses such as vision, touch, and the ability to sense temperature.
- Some are even capable of simple decision making, and current robotics research is geared toward devising robots with a degree of self-sufficiency that will permit mobility and decision-making in an unstructured environment

Interface College of Computer Applications (ICCA)

Robot Components and Mechanics:

- Actuators: Motors and mechanisms that enable movement and manipulation.
- Sensors: Devices for gathering data about the robot's environment (e.g., cameras, LiDAR, proximity sensors).
- End Effectors: Tools or grippers attached to robots for interacting with objects.

Types of Robots:

- Industrial Robots: Used in manufacturing for tasks like assembly, welding, and packaging.
- Service Robots: Designed to assist or entertain humans in environments such as homes, hospitals, and public spaces.
- Mobile Robots: Autonomous or semi-autonomous robots that can navigate and operate in different environments.

Applications of Robotics

1. Manufacturing and Industrial Automation:
   - Assembly Lines: Robots perform repetitive tasks such as welding, painting, and assembly in automotive and electronics manufacturing.
   - Material Handling: Autonomous mobile robots (AMRs) transport materials and goods within warehouses and distribution centers.
   - Quality Control: Robots inspect products for defects, ensuring consistent quality and reducing errors.
2. Healthcare:
   - Surgical Robotics: Assist surgeons in performing minimally invasive surgeries with precision and dexterity, reducing recovery times and improving outcomes.
   - Medical Robotics: Aid in rehabilitation therapies, prosthetics, and exoskeletons to assist patients with mobility impairments.
3. Logistics and Warehousing:
   - Order Fulfillment: AMRs and robotic arms pick, pack, and sort orders in fulfillment centers, increasing efficiency in e-commerce operations.
   - Inventory Management: Drones and robots equipped with RFID or vision systems conduct inventory counts and manage stock levels.
4. Agriculture:
   - Precision Farming: Robots and drones monitor crops, apply fertilizers and pesticides with precision, and harvest crops autonomously.
   - Livestock Management: Robots feed, monitor, and sort livestock in farms, improving animal welfare and farm productivity.
5. Service and Hospitality:
   - Cleaning Robots: Autonomous vacuum cleaners and floor scrubbers clean large spaces such as airports, malls, and offices.
   - Customer Service Robots: Guide visitors in museums, airports, and shopping malls, providing information and assistance.
6. Space Exploration:
   - Planetary Exploration: Rovers like NASA's Curiosity explore Mars, conducting experiments and collecting samples.
   - Satellite Servicing: Robots repair, refuel, and maintain satellites in orbit, extending their operational lifespans.
7. Defense and Security:
   - Unmanned Aerial Vehicles (UAVs): Drones perform reconnaissance, surveillance, and monitoring missions in military and law enforcement operations.
   - Bomb Disposal Robots: Remote-controlled robots handle and disarm explosive devices safely.
8. Education and Research:
   - Educational Robotics: Programmable kits and platforms introduce students to robotics and STEM concepts, fostering creativity and problem-solving skills.
   - Research Robots: Assist scientists in exploring underwater environments, studying marine life, and conducting experiments in laboratories.
9. Entertainment and Media:
   - Film Production: Robots equipped with cameras capture complex shots and sequences in movies and television productions.

Interface College of Computer Applications (ICCA)

- Theme Parks: Robots entertain visitors with interactive performances, rides, and attractions.

10. Environmental Monitoring and Disaster Response:
    - Search and Rescue Robots: Robots navigate disaster zones to locate survivors and deliver supplies in hazardous conditions.
    - Environmental Monitoring: Drones and underwater robots monitor ecosystems, track wildlife, and assess environmental health.

Interface College of Computer Applications (ICCA)