STUDY MATERIAL FOR B.Sc., COMPUTER SCIENCE WITH ARTIFICIAL INTELLIGENCE

ARTIFICIAL INTELLIGENCE AND KNOWLEDGE REPRESENTATION

SEMESTER – II

ACADEMIC YEAR 2025-26

PREPARED BY

COMPUTER SCIENCE DEPARTMENT

INDEX

UNIT - I

Introduction

- Artificial Intelligence is concerned with the design of intelligence in an artificial device. The term was coined by John McCarthy in 1956.
- Intelligence is the ability to acquire, understand and apply the knowledge to achieve goals in the world.
- AI is the study of the mental faculties through the use of computational models
- AI is the study of intellectual/mental processes as computational processes.
- AI program will demonstrate a high level of intelligence to a degree that equals or exceeds the intelligence required of a human in performing some task.
- AI is unique, sharing borders with Mathematics, Computer Science, Philosophy, Psychology, Biology, Cognitive Science and many others.
- Although there is no clear definition of AI or even Intelligence, it can be described as an attempt to build machines that like humans can think and act, able to learn and use knowledge to solve problems on their own.

History of AI:

Important research that laid the groundwork for AI:

- In 1931, Goedel layed the foundation of Theoretical Computer Science1920-30s:
- He published the first universal formal language and showed that math itself is either flawed or allows for unprovable but true statements.
- In 1936, Turing reformulated Goedel's result and church's extension thereof.
- In 1956, John McCarthy coined the term "Artificial Intelligence" as the topic of the
- Dartmouth Conference, the first conference devoted to the subject.
- In 1957, The General Problem Solver (GPS) demonstrated by Newell, Shaw & SimoN
- In 1958, John McCarthy (MIT) invented the Lisp language.
- In 1959, Arthur Samuel (IBM) wrote the first game-playing program, for checkers, to achieve sufficient skill to challenge a world champion.
- In 1963, Ivan Sutherland's MIT dissertation on Sketchpad introduced the idea of interactive graphics into computing.

- In 1966, Ross Quillian (PhD dissertation, Carnegie Inst. of Technology; now CMU) demonstrated semantic nets
- In 1967, Dendral program (Edward Feigenbaum, Joshua Lederberg, Bruce Buchanan, Georgia Sutherland at Stanford) demonstrated to interpret mass spectra on organic chemical compounds. First successful knowledge-based program for scientific reasoning.
- In 1967, Doug Engelbart invented the mouse at SRI
- In 1968, Marvin Minsky & Seymour Papert publish Perceptrons, demonstrating limits of simple neural nets.
- In 1972, Prolog developed by Alain Colmerauer.
- In Mid 80's, Neural Networks become widely used with the Backpropagation algorithm (first described by Werbos in 1974).
- 1990, Major advances in all areas of AI, with significant demonstrations in machine learning, intelligent tutoring, case-based reasoning, multi-agent planning, scheduling, uncertain reasoning, data mining, natural language understanding and translation, vision, virtual reality, games, and other topics.
- In 1997, Deep Blue beats the World Chess Champion Kasparov
- In 2002,iRobot, founded by researchers at the MIT Artificial Intelligence Lab, introduced Roomba, a vacuum cleaning robot. By 2006, two million had been sold.

- CS-based AI started with "Dartmouth Conference" in 1956
- Attendees
  - John McCarthy
    - LISP, application of logic to reasoning
  - Marvin Minsky
    - Popularized neural networks
    - Slots and frames
    - The Society of the Mind
  - Claude Shannon
    - Computer checkers
    - Information theory
    - Open-loop 5-ball juggling
  - Allen Newell and Herb Simon
    - General Problem Solver

What is Artificial Intelligence?

- It is a branch of Computer Science that pursues creating the computers or machines as intelligent as human beings.
- It is the science and engineering of making intelligent machines, especially intelligent computer programs.
- It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable

Definition: Artificial Intelligence is the study of how to make computers do things, which, at the moment, people do better.

According to the father of Artificial Intelligence, John McCarthy, it is "The science and engineering of making intelligent machines, especially intelligent computer programs".

Artificial Intelligence is a way of making a computer, a computer-controlled robot, or a software think intelligently, in the similar manner the intelligent humans think.

AI is accomplished by studying how human brain thinks and how humans learn, decide, and work while trying to solve a problem, and then using the outcomes of this study as a basis of developing intelligent software and systems.

It has gained prominence recently due, in part, to big data, or the increase in speed, size and variety of data businesses are now collecting.

AI can perform tasks such as identifying patterns in the data more efficiently than humans, enabling businesses to gain more insight out of their data.

From a business perspective AI is a set of very powerful tools, and methodologies for using those tools to solve business problems.

From a programming perspective, AI includes the study of symbolic programming, problem solving, and search.

AI Vocabulary

Intelligence relates to tasks involving higher mental processes, e.g. creativity, solving problems, pattern recognition, classification, learning, induction, deduction, building analogies, optimization, language processing, knowledge and many more. Intelligence is the computational part of the ability to achieve goals.

Intelligent behaviour is depicted by perceiving one's environment, acting in complex environments, learning and understanding from experience, reasoning to solve problems and discover hidden knowledge, applying knowledge successfully in new situations, thinking abstractly, using analogies, communicating with others and more.

Science based goals of AI pertain to developing concepts, mechanisms and understanding biological intelligent behaviour. The emphasis is on understanding intelligent behaviour.

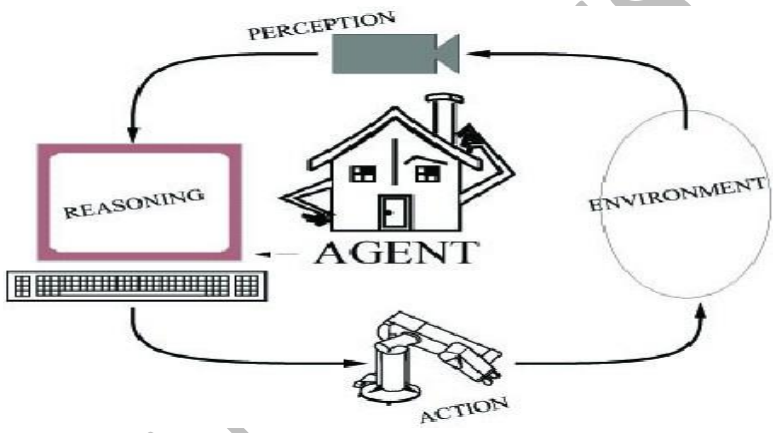
Engineering based goals of AI relate to developing concepts, theory and practice of building intelligent machines. The emphasis is on system building.

AI Techniques depict how we represent, manipulate and reason with knowledge in order to solve problems. Knowledge is a collection of 'facts'. To manipulate these facts by a program, a suitable representation is required. A good representation facilitates problem solving.

Learning means that programs learn from what facts or behaviour can represent. Learning denotes changes in the systems that are adaptive in other words, it enables the system to do the same task(s) more efficiently next time.

Applications of AI refers to problem solving, search and control strategies, speech recognition, natural language understanding, computer vision, expert systems, etc.

Components of an AI System



An agent perceives its environment through sensors and acts on the environment through actuators.

Human: sensors are eyes, ears, actuators (effectors) are hands, legs, mouth.

Robot: sensors are cameras, sonar, lasers, ladar, bump, effectors are grippers, manipulators, motors

The agent's behavior is described by its function that maps percept to action.

Problems of AI:

Intelligence does not imply perfect understanding; every intelligent being has limited perception, memory and computation. Many points on the spectrum of intelligence versus cost are viable, from insects to humans. AI seeks to understand the computations required from intelligent behaviour and to produce computer systems that exhibit intelligence. Aspects of intelligence

studied by AI include perception, communicational using human languages, reasoning, planning, learning and memory.

The following questions are to be considered before we can step forward:

1. What are the underlying assumptions about intelligence?
2. What kinds of techniques will be useful for solving AI problems?
3. At what level human intelligence can be modelled?
4. When will it be realized when an intelligent program has been built?

Applications of AI

AI has applications in all fields of human study, such as finance and economics, environmental engineering, chemistry, computer science, and so on.

Some of the applications of AI are listed below:

- Perception
    - Machine vision
    - Speech understanding
    - Touch (tactile or haptic) sensation
- Robotics
- Natural Language Processing
    - Natural Language Understanding
    - Speech Understanding
    - Language Generation
    - Machine Translation
- Planning
- Expert Systems
- Machine Learning
- Theorem Proving
- Symbolic Mathematics
- Game Playing

Foundations of Artificial Intelligence:

- **Philosophy**

  e.g., foundational issues (can a machine think?), issues of knowledge and believe, mutual knowledge

- **Psychology and Cognitive Science**

  e.g., problem solving skills

- **Neuro-Science**

  e.g., brain architecture

- **Computer Science And Engineering**

  e.g., complexity theory, algorithms, logic and inference, programming languages, and system building.

- **Mathematics and Physics**

  e.g., statistical modeling, continuous mathematics,

Statistical Physics, and Complex Systems. Application of AI:

AI algorithms have attracted close attention of researchers and have also been applied successfully to solve problems in engineering. Nevertheless, for large and complex problems, AI algorithms consume considerable computation time due to stochastic feature of the search approaches

1. Business; financial strategies
2. Engineering: check design, offer suggestions to create new product, expert systems for all engineering problems
3. Manufacturing: assembly, inspection and maintenance
4. Medicine: monitoring, diagnosing
5. Education: in teaching
6. Fraud detection
7. Object identification
8. Information retrieval
9. Space shuttle scheduling

Building AI Systems:

Perception

Intelligent biological systems are physically embodied in the world and experience the world through their sensors (senses). For an autonomous vehicle, input might be images from a camera and range information from a rangefinder. For a medical diagnosis system, perception is the set of symptoms and test results that have been obtained and input to the system manually.

Reasoning

Inference, decision-making, classification from what is sensed and what the internal "model" is of the world. Might be a neural network, logical deduction system, Hidden Markov Model induction, heuristic searching a problem space, Bayes Network inference, genetic algorithms, etc.

Includes areas of knowledge representation, problem solving, decision theory, planning, game theory, machine learning, uncertainty reasoning, etc.

Action

Biological systems interact within their environment by actuation, speech, etc. All behavior is cantered around actions in the world. Examples include controlling the steering of a Mars rover or autonomous vehicle, or suggesting tests and making diagnoses for a medical diagnosis system. Includes areas of robot actuation, natural language generation, and speech synthesis.

The definitions of AI:

a) "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)	b) "The study of mental faculties through the use of computational models" (Charniak and McDermott, 1985) "The study of the computations that make it possible to perceive, reason, and act" (Winston, 1992)
c) "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, 1 99 1)	d) "A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes" (Schalkoff, 1 990) "The branch of computer science that is concerned with the automation of intelligent behavior" (Luger and Stubblefield, 1993)

The definitions on the top, (a) and (b) are concerned with reasoning, whereas those on the bottom, (c) and (d) address behavior. The definitions on the left, (a) and (c) measure success in terms of human performance, and those on the right, (b) and (d) measure the ideal concept of intelligence called rationality.

Intelligent Systems:

In order to design intelligent systems, it is important to categorize them into four categories (Luger and Stubberfield 1993), (Russell and Norvig, 2003)

1. Systems that think like humans
2. Systems that think rationally
3. Systems that behave like humans
4. Systems that behave rationally

	Human- Like	**Rationally**
Think:	Cognitive Science Approach "Machines that think like humans"	Laws of thought Approach " Machines that think Rationally"
Act:	Turing Test Approach "Machines that behave like humans"	Rational Agent Approach "Machines that behave Rationally"

Scientific Goal: To determine which ideas about knowledge representation, learning, rule systems search, and so on, explain various sorts of real intelligence.

Engineering Goal: To solve real world problems using AI techniques such as Knowledge representation, learning, rule systems, search, and so on.

Traditionally, computer scientists and engineers have been more interested in the engineering goal, while psychologists, philosophers and cognitive scientists have been more interested in the scientific goal.

Cognitive Science: Think Human-Like

- Requires a model for human cognition. Precise enough models allow simulation by computers.
- Focus is not just on behavior and I/O, but looks like reasoning process.
- Goal is not just to produce human-like behavior but to produce a sequence of steps of the reasoning process, similar to the steps followed by a human in solving the same task.

Laws of thought: Think Rationally

- The study of mental faculties through the use of computational models; that it is, the study of computations that make it possible to perceive reason and act.
- Focus is on inference mechanisms that are probably correct and guarantee an optimal solution.
- Goal is to formalize the reasoning process as a system of logical rules and procedures of inference.
- Develop systems of representation to allow inferences to be like "Socrates is a man. All men are mortal. Therefore Socrates is mortal"

Turing Test: Act Human-Like

- The art of creating machines that perform functions requiring intelligence when performed by people; that it is the study of, how to make computers do things which, at the moment, people do better.
- Focus is on action, and not intelligent behavior centered around the representation of the world

Example: Turing Test

- 3 rooms contain: a person, a computer and an interrogator.
- The interrogator can communicate with the other 2 by teletype (to avoid the machine imitate the appearance of voice of the person)
- The interrogator tries to determine which the person is and which the machine is.
- The machine tries to fool the interrogator to believe that it is the human, and the person also tries to convince the interrogator that it is the human.
- If the machine succeeds in fooling the interrogator, then conclude that the machine is intelligent.

Rational agent: Act Rationally

- Tries to explain and emulate intelligent behavior in terms of computational process; that it is concerned with the automation of the intelligence.
- Focus is on systems that act sufficiently if not optimally in all situations.
- Goal is to develop systems that are rational and sufficient

Intelligent Agent's

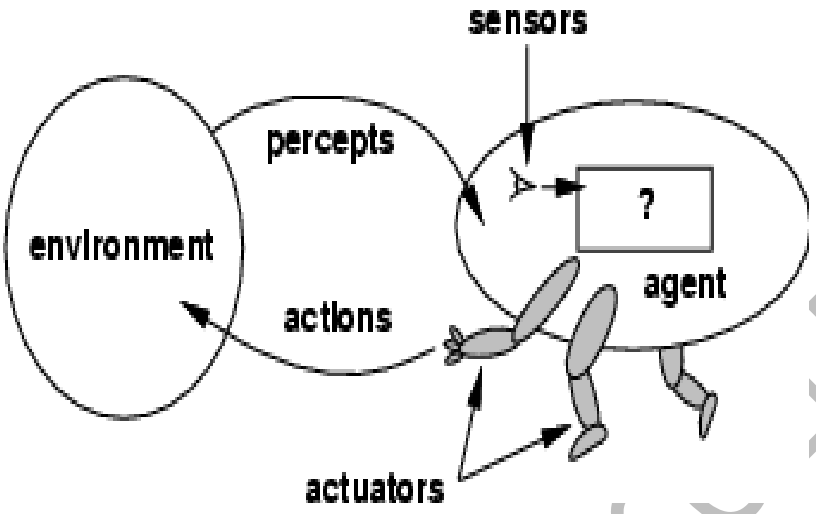


Fig : Agents and Environments

Agent:

An Agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.

- A human agent has eyes, ears, and other organs for sensors and hands, legs, mouth, and other body parts foractuators.
- A robotic agent might have cameras and infrared range finders for sensors and various motors foractuators.
- A software agent receives keystrokes, file contents, and network packets as sensory inputs and acts on the environment by displaying on the screen, writing files, and sending network packets.

Percept:

We use the term percept to refer to the agent's perceptual inputs at any given instant.

PerceptSequence:

An agent's percept sequence is the complete history of everything the agent has ever perceived.

Agent function:

Mathematically speaking, we say that an agent's behavior is described by the agent function that maps any given percept sequence to an action.

Agent program

Internally, the agent function for an artificial agent will be implemented by an agent program. It is important to keep these two ideas distinct. The agent function is an abstract mathematical description; the agent program is a concrete implementation, running on the agent architecture.

PEAS

- Use PEAS to describe task
  - Performance measure
  - Environment
  - Actuators
  - Sensors
- Example: Taxi driver
  - Performance measure: safe, fast, comfortable (maximize profits)
  - Environment: roads, other traffic, pedestrians, customers
  - Actuators: steering, accelerator, brake, signal, horn
  - Sensors: cameras, sonar, speedometer, GPS, odometer, accelerometer, engine sensors

Agent types

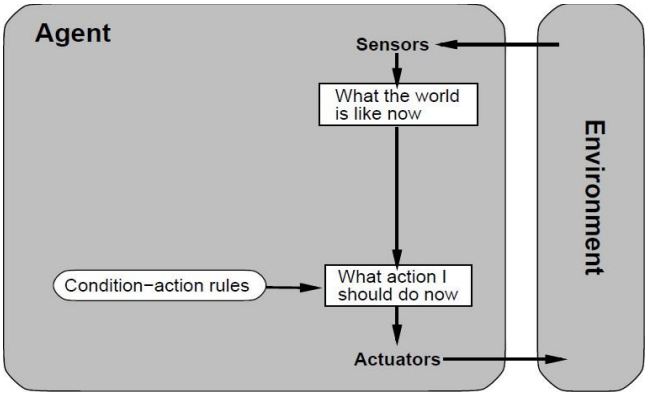
5 basic types in order of increasing generality:

1. Simple reflex agents
2. Model-based reflex agents with state
3. Goal-based agents
4. Utility-based agents
5. Learning agents

All these can be turned into learning agents

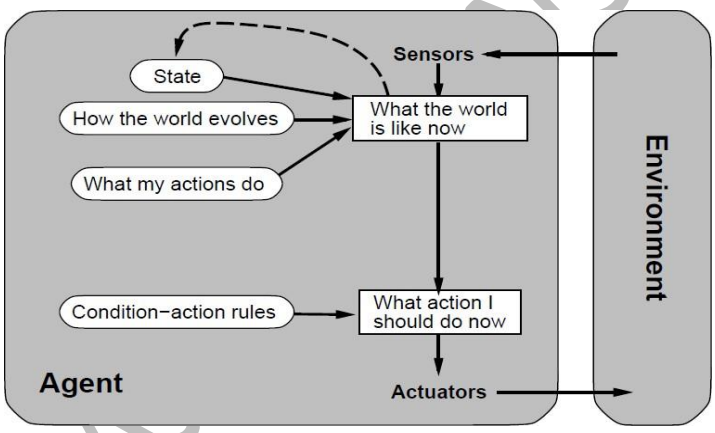
1. Simple reflex agents

These agents perform actions using the current percept, rather than the percept history. The condition-action rule is used as the basis for the agent function. In this category, a fully observable environment is ideal for the success of the agent function.



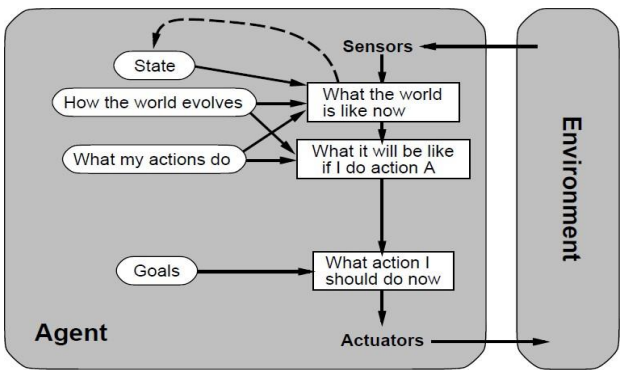
2. Model-based reflex agents with state

Unlike simple reflex agents, model-based reflex agents consider the percept history in their actions. The agent function can still work well even in an environment that is not fully observable. These agents use an internal model that determines the percept history and effect of actions. They reflect on certain aspects of the present state that have been unobserved



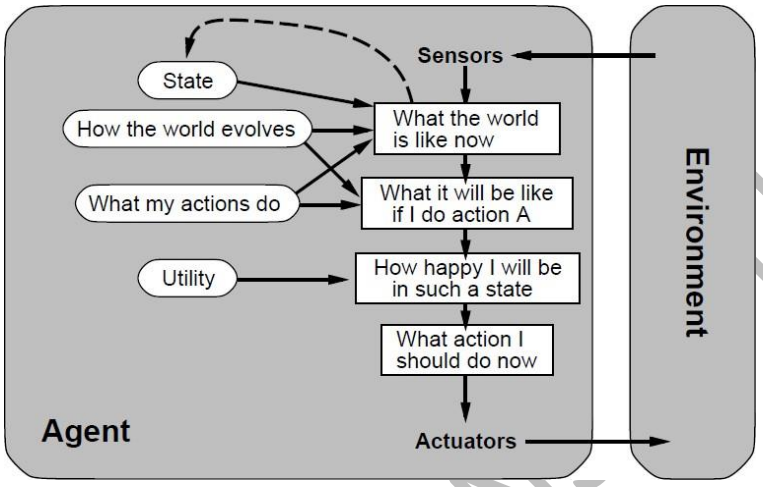
3. Goal-based agents

These agents have higher capabilities than model-based reflex agents. Goal-based agents use goal information to describe desirable capabilities. This allows them to choose among various possibilities. These agents select the best action that enhances the attainment of the goal.



4. Utility-based agents

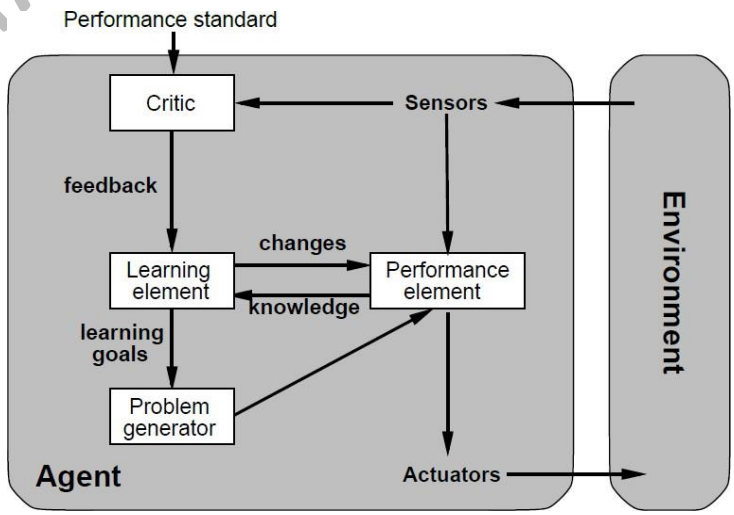
These agents make choices based on utility. They are more advanced than goal-based agents because of an extra component of utility measurement. Using a utility function, a state is mapped against a certain measure of utility. A rational agent selects the action that optimizes the expected utility of the outcome.



5. Learning Agents

These are agents that have the capability of learning from their previous experience. Learning agents have the following elements.

- **The learning element:** This element enables learning agents to learn from previous experiences.
- **The critic:** It provides feedback on how the agent is doing.
- **The performance element:** This element decides on the external action that needs to be taken.
- **The problem generator:** This acts as a feedback agent that performs certain tasks such as making suggestions (new) and keeping history.



Environment Properties

- Fully observable vs. partially observable
- Deterministic vs. stochastic / strategic
- Episodic vs. sequential
- Static vs. dynamic
- Discrete vs. continuous
- Single agent vs. multiagent

Environment I

Fully observable (accessible) vs. partially observable (inaccessible):

- Fully observable if agents sensors detect all aspects of environment relevant to choice of action
- Could be partially observable due to noisy, inaccurate or missing sensors, or inability to measure everything that is needed
- Model can keep track of what was sensed previously, cannot be sensed now, but is probably still true.
- Often, if other agents are involved, their intentions are not observable, but their actions are
- E.g
    - chess – the board is fully observable, as are opponent's moves.
    - Driving – what is around the next bend is not observable (yet).

Environment II

Deterministic vs. stochastic (non-deterministic):

- Deterministic = the next state of the environment is completely predictable from the current state and the action executed by the agent
- Stochastic = the next state has some uncertainty associated with it
- Uncertainty could come from randomness, lack of a good environment model, or lack of complete sensor coverage
- Strategic environment if the environment is deterministic except for the actions of other agents

Examples:

Non-deterministic environment: physical

world: Robot on Mars Deterministic

environment: Tic Tac Toe game

Environment III

Episodic vs. sequential:

The agent's experience is divided into atomic "episodes" (each episode consists of the agent perceiving and then performing a single action) and the choice of action in each episode depends only on the episode itself

Sequential if current decisions affect future decisions, or rely on previous ones

Examples of episodic are expert advice systems – an episode is a single question and answer

Most environments (and agents) are sequential

Many are both – a number of episodes containing a number of sequential steps to a conclusion

Examples:

Episodic environment: mail sorting system

Non-episodic environment: chess game

Environment IV

Discrete vs. continuous:

Discrete = time moves in fixed steps, usually with one measurement per step (and perhaps one action, but could be no action). E.g. a game of chess

Continuous = Signals constantly coming into sensors, actions continually changing. E.g. driving a car

Environment V

Static vs. dynamic:

Dynamic if the environment may change over time.

Static if nothing (other than the agent) in the environment changes

Other agents in an environment make it dynamic

The goal might also change over time

Not dynamic if the agent moves from one part of an environment to another, though it has a very similar effect

E.g. – Playing football, other players make it dynamic, mowing a lawn is static (unless there is a cat...), expert systems usually static (unless knowledge changes)

Environment VI

Single agent vs. multi agent:

An agent operating by itself in an environment is single agent!

Multi agent is when other agents are present!

A strict definition of another agent is anything that changes from step to step.

A stronger definition is that it must sense and act

Competitive or co-operative Multi-agent environments

Human users are an example of another agent in a system

E.g. Other players in a football team (or opposing team), wind and waves in a sailing agent, other cars in a taxi drive

Problem Formulation

The reflex agents are known as the simplest agents because they directly map states into actions. Unfortunately, these agents fail to operate in an environment where the mapping is too large to store and learn. Goal-based agent, on the other hand, considers future actions and the desired outcomes.

Here, we will discuss one type of goal-based agent known as a problem-solving agent, which uses atomic representation with no internal states visible to the problem-solving algorithms.

Problem-solving agent

- The problem-solving agent performs precisely by defining problems and its several solutions.
- According to psychology, “a problem-solving refers to a state where we wish to reach to a definite goal from a present state or condition.”
- According to computer science, a problem-solving is a part of artificial intelligence which encompasses a number of techniques such as algorithms, heuristics to solve a problem.

Therefore, a problem-solving agent is a goal-driven agent and focuses on satisfying the goal.

2 Steps performed by Problem-solving agent

- Goal Formulation: It is the first and simplest step in problem-solving. It organizes the steps/sequence required to formulate one goal out of multiple goals as well as actions to achieve that goal. Goal formulation is based on the current situation and the agent's performance measure (discussed below).
- Problem Formulation: It is the most important step of problem-solving which decides what actions should be taken to achieve the formulated goal.

There are following five components involved in problem formulation:

1. **Initial State:** It is the starting state or initial step of the agent towards its goal.
2. **Actions:** It is the description of the possible actions available to the agent.
3. **Transition Model:** It describes what each action does.
4. **Goal Test:** It determines if the given state is a goal state.
5. **Path cost:** It assigns a numeric cost to each path that follows the goal.

The problem- solving agent selects a cost function, which reflects its performance measure.

Remember, an optimal solution has the lowest path cost among all the solutions.

Note: Initial state, actions, and transition model together define the state-space of the problem implicitly. State-space of a problem is a set of all states which can be reached from the initial state followed by any sequence of actions. The state-space forms a directed map or graph where nodes are the states, links between the nodes are actions, and the path is a sequence of states connected by the sequence of actions.

- **Search:** It identifies all the best possible sequence of actions to reach the goal state from the current state. It takes a problem as an input and returns solution as its output.
- **Solution:** It finds the best algorithm out of various algorithms, which may be proven as the best optimal solution.
- **Execution:** It executes the best optimal solution from the searching algorithms to reach the goal state from the current state.

Example Problems

Basically, there are two types of problem approaches:

- **Toy Problem:** It is a concise and exact description of the problem which is used by the researchers to compare the performance of algorithms.

- **Real-world Problem:** It is real-world based problems which require solutions. Unlike a toy problem, it does not depend on descriptions, but we can have a general formulation of the problem.

Searching Solutions:

To build a system to solve a problem:

1. Define the problem precisely
2. Analyze the problem
3. Isolate and represent the task knowledge that is necessary to solve the problem
4. Choose the best problem-solving techniques and apply it to the particular problem.

State Space Search

Defining the problem as State Space Search:

The state space representation forms the basis of most of the AI methods.

- Formulate a problem as a state space search by showing the legal problem states, the legal operators, and the initial and goal states.
- A state is defined by the specification of the values of all attributes of interest in the world
- An operator changes one state into the other; it has a precondition which is the value of certain attributes prior to the application of the operator, and a set of effects, which are the attributes altered by the operator
- The initial state is where you start
- The goal state is the partial description of the solution

Formal Description of the problem:

1. Define a state space that contains all the possible configurations of the relevant objects.
2. Specify one or more states within that space that describe possible situations from which the problem solving process may start (initial state)
3. Specify one or more states that would be acceptable as solutions to the problem. (goal states)
4. Specify a set of rules that describe the actions (operations) available

State-Space Problem Formulation:

Example: A problem is defined by four items:

1. **initial state e.g., "at Arad—**

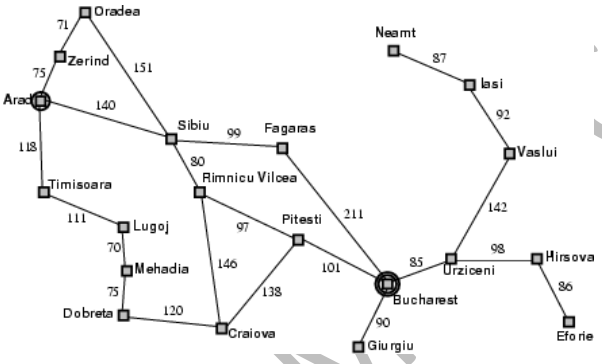
2. **actions or successor function** : S(x) = set of action–state pairs e.g., S(Arad) = {<Arad Zerind, Zerind>, … }
3. **goal test (or set of goal states)**

   e.g., x = "at Bucharest‖, Checkmate(x)
4. **path cost (additive)**

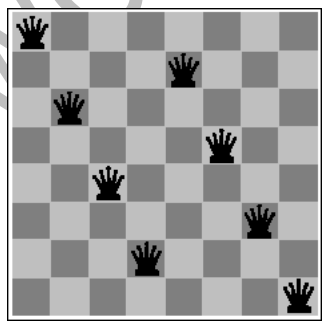
   e.g., sum of distances, number of actions executed, etc.

   c(x,a,y) is the step cost, assumed to be ≥ 0

A solution is a sequence of actions leading from the initial state to a goal state



Example: 8-queens problem



1. **Initial State:** Any arrangement of 0 to 8 queens on board.
2. **Operators:** add a queen to any square.
3. **Goal Test:** 8 queens on board, none attacked.
4. **Path cost:** not applicable or Zero (because only the final state counts, search cost might be of interest).

State Spaces versus Search Trees:

State Space

- Set of valid states for a problem
- Linked by operators
- e.g., 20 valid states (cities) in the Romanian travel problem

Search Tree

- Root node = initial state
- Child nodes = states that can be visited from parent
- Note that the depth of the tree can be infinite E.g., via repeated states
- Partial search tree

  Portion of tree that has been expanded so far
- Fringe

  Leaves of partial search tree, candidates for expansion

Search trees = data structure to search state-space

Properties of Search Algorithms

Which search algorithm one should use will generally depend on the problem domain. There are four important factors to consider:

1. **Completeness** – Is a solution guaranteed to be found if at least one solution exists?
2. **Optimality** – Is the solution found guaranteed to be the best (or lowest cost) solution if there exists more than one solution?
3. **Time Complexity** – The upper bound on the time required to find a solution, as a function of the complexity of the problem.
4. **Space Complexity** – The upper bound on the storage space (memory) required at any point during the search, as a function of the complexity of the problem.

UNIT - II

Searching Techniques:

Search Algorithms in Artificial Intelligence

Search algorithms are one of the most important areas of Artificial Intelligence.

Problem-solving agents:

In Artificial Intelligence, Search techniques are universal problem-solving methods. Rational agents or Problem-solving agents in AI mostly used these search strategies or algorithms to solve a specific problem and provide the best result. Problem-solving agents are the goal-based agents and use atomic representation.

There are various problem-solving search algorithms.

Search Algorithm Terminologies:

- **Search:** Searching is a step by step procedure to solve a search-problem in a given search space. A search problem can have three main factors:
    1. **Search Space:** Search space represents a set of possible solutions, which a system may have.
    2. **Start State:** It is a state from where agent begins the search.
    3. **Goal test:** It is a function which observe the current state and returns whether the goal state is achieved or not.
- **Search tree:** A tree representation of search problem is called Search tree. The root of the search tree is the root node which is corresponding to the initial state.
- **Actions:** It gives the description of all the available actions to the agent.
- **Transition model:** A description of what each action do, can be represented as a transition model.
- **Path Cost:** It is a function which assigns a numeric cost to each path.
- **Solution:** It is an action sequence which leads from the start node to the goal node.
- **Optimal Solution:** If a solution has the lowest cost among all solutions.

Properties of Search Algorithms:

Following are the four essential properties of search algorithms to compare the efficiency of these algorithms:

Completeness: A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input.

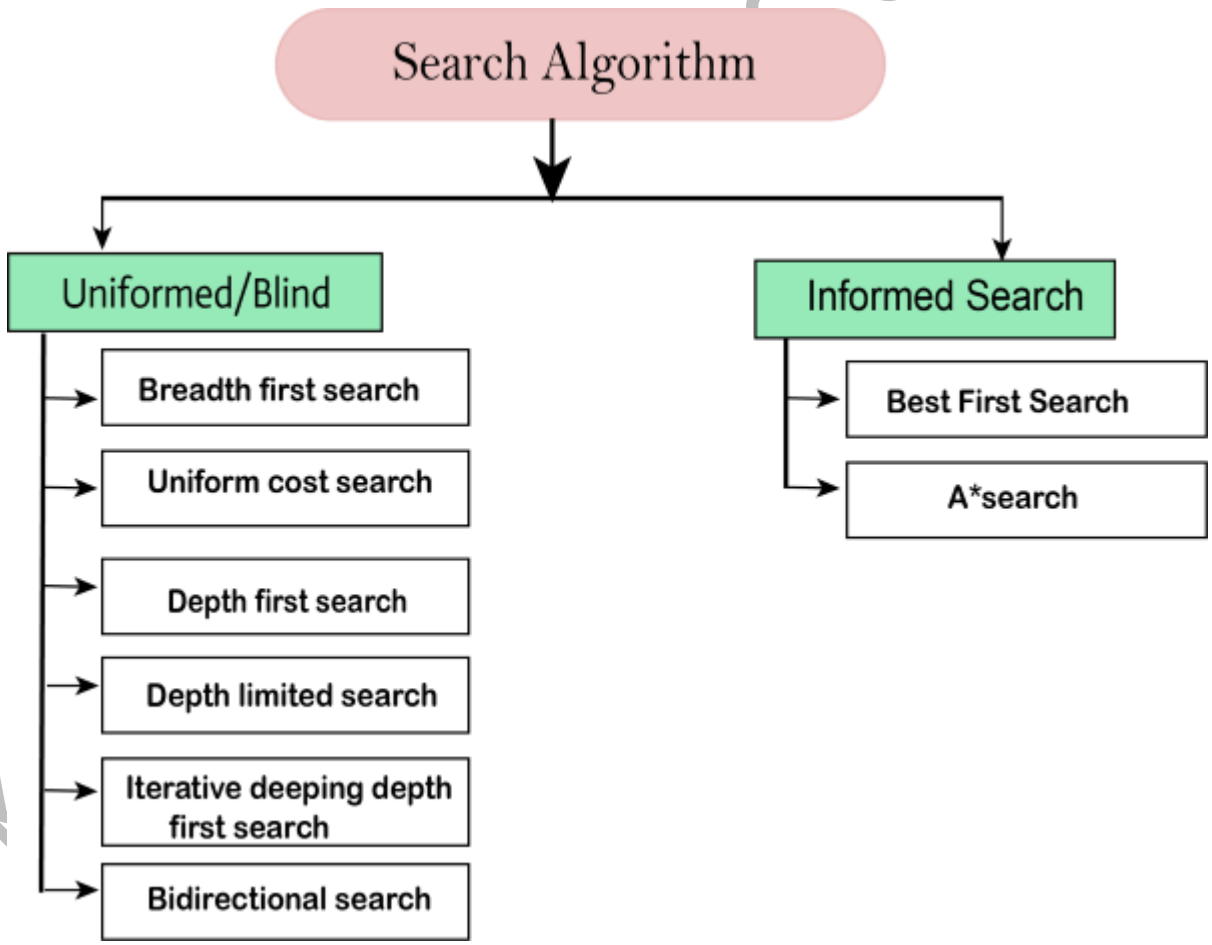
Optimality: If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for is said to be an optimal solution.

Time Complexity: Time complexity is a measure of time for an algorithm to complete its task.

Space Complexity: It is the maximum storage space required at any point during the search, as the complexity of the problem.

Types of search algorithms

Based on the search problems we can classify the search algorithms into uninformed (Blind search) search and informed search (Heuristic search) algorithms.



Uninformed/Blind Search:

The uninformed search does not contain any domain knowledge such as closeness, the location of the goal. It operates in a brute-force way as it only includes information about how to traverse the tree and how to identify leaf and goal nodes.

Uninformed search applies a way in which search tree is searched without any information about the search space like initial state operators and test for the goal, so it is also called blind search. It examines each node of the tree until it achieves the goal node.

It can be divided into five main types:

- Breadth-first search
- Uniform cost search
- Depth-first search
- Iterative deepening depth-first search
- Bidirectional Search

Informed Search

Informed search algorithms use domain knowledge. In an informed search, problem information is available which can guide the search. Informed search strategies can find a solution more efficiently than an uninformed search strategy. Informed search is also called a Heuristic search.

A heuristic is a way which might not always be guaranteed for best solutions but guaranteed to find a good solution in reasonable time.

Informed search can solve much complex problem which could not be solved in another way. An example of informed search algorithms is a traveling salesman problem.

1. Greedy Search
2. A* Search

Difference between Informed and Uninformed Search

Informed Search	Uninformed Search
It uses knowledge for the searching process.	It doesn't use knowledge for searching process.
It finds solution more quickly.	It finds solution slow as compared to informed search.
It may or may not be complete.	It is always complete.
Cost is low.	Cost is high.
It consumes less time.	It consumes moderate time.

It provides the direction regarding the solution.	No suggestion is given regarding the solution in it.
It is less lengthy while implementation.	It is more lengthy while implementation.
Greedy Search, A* Search, Graph Search	BFS,DFS

Uninformed Search Algorithms

Uninformed search is a class of general-purpose search algorithms which operates in brute force-way. Uninformed search algorithms do not have additional information about state or search space other than how to traverse the tree, so it is also called blind search.

Following are the various types of uninformed search algorithms:

1. Breadth-first Search
2. Depth-first Search
3. Depth-limited Search
4. Iterative deepening depth-first search
5. Uniform cost search
6. Bidirectional Search

1. Breadth-first Search:

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.
- The breadth-first search algorithm is an example of a general-graph search algorithm.
- Breadth-first search implemented using FIFO queue data structure.

Advantages:

- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

Disadvantages:

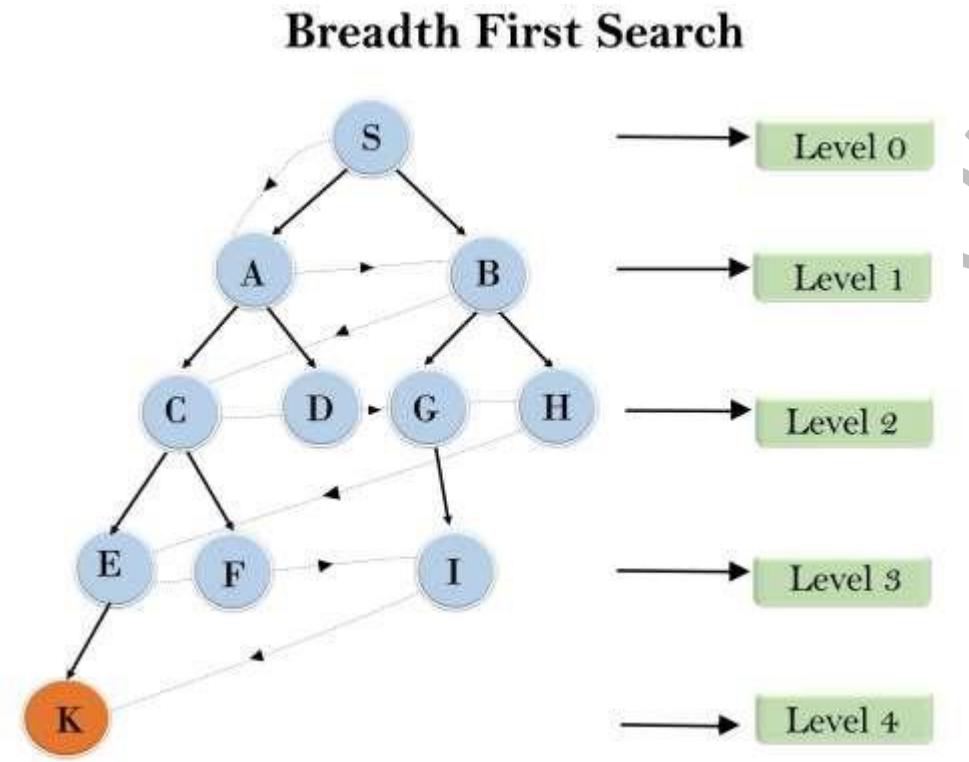
- It requires lots of memory since each level of the tree must be saved into memory to expand the next level.

- BFS needs lots of time if the solution is far away from the root node.

Example:

In the below tree structure, we have shown the traversing of the tree using BFS algorithm from the root node S to goal node K. BFS search algorithm traverse in layers, so it will follow the path which is shown by the dotted arrow, and the traversed path will be:

S---> A--->B---->C--->D---->G--->H--->E---->F---->I >K



Time Complexity: Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.

T (b) = 1+b2+b3+. + bd= O (bd)

Space Complexity: Space complexity of BFS algorithm is given by the Memory size of frontier which is O(bd).

Completeness: BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.

Optimality: BFS is optimal if path cost is a non-decreasing function of the depth of the node.

2. Depth-first Search

- Depth-first search isa recursive algorithm for traversing a tree or graph data structure.

- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a stack data structure for its implementation.
- The process of the DFS algorithm is similar to the BFS algorithm.

Note: Backtracking is an algorithm technique for finding all possible solutions using recursion.

Advantage:

- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

Disadvantage:

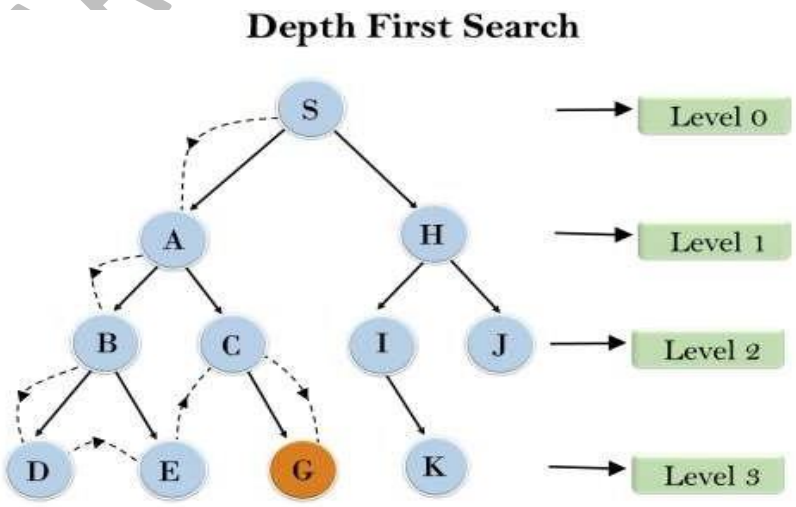
- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

Example:

In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:

Root node--->Left node-----> right node.

It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still goal node is not found. After backtracking it will traverse node C and then G, and here it will terminate as it found goal node.



Completeness: DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.

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(nm)

Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)

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).

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.

3. Depth-Limited Search Algorithm:

A depth-limited search algorithm is similar to depth-first search with a predetermined limit. Depth-limited search can solve the drawback of the infinite path in the Depth-first search. In this algorithm, the node at the depth limit will treat as it has no successor nodes further.

Depth-limited search can be terminated with two Conditions of failure:

- Standard failure value: It indicates that problem does not have any solution.
- Cutoff failure value: It defines no solution for the problem within a given depth limit.

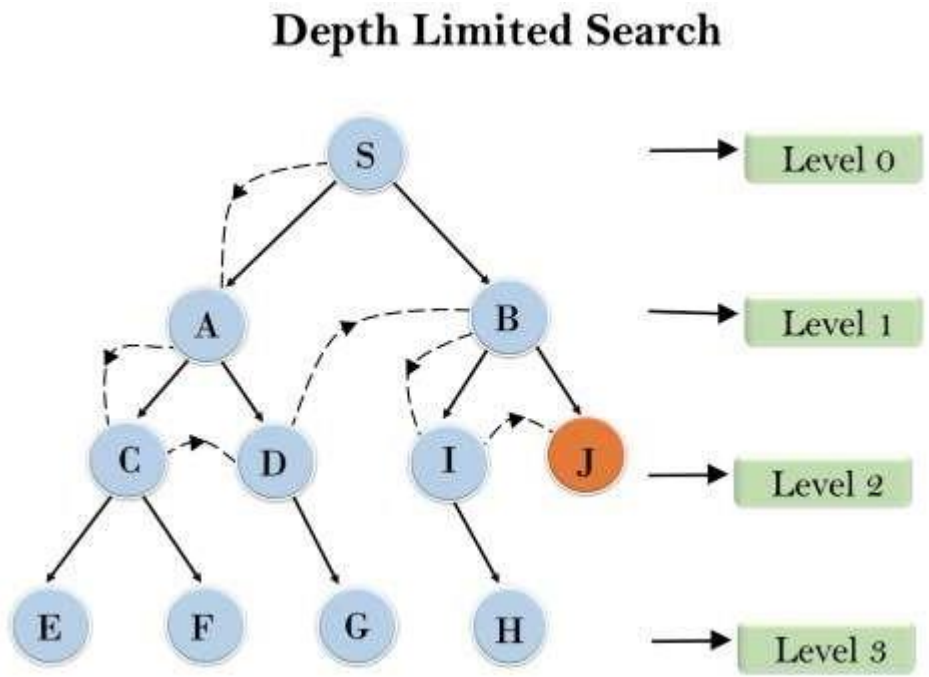
Advantages:

- Depth-limited search is Memory efficient.

Disadvantages:

- Depth-limited search also has a disadvantage of incompleteness.
- It may not be optimal if the problem has more than one solution.

Example:



Completeness: DLS search algorithm is complete if the solution is above the depth-limit.

Time Complexity: Time complexity of DLS algorithm is O(bℓ). Space Complexity: Space complexity of DLS algorithm is O(b×ℓ).

Optimal: Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if ℓ>d.

4. Uniform-cost Search Algorithm:

Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge. The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost. Uniform-cost search expands nodes according to their path costs form the root node. It can be used to solve any graph/tree where the optimal cost is in demand. 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.

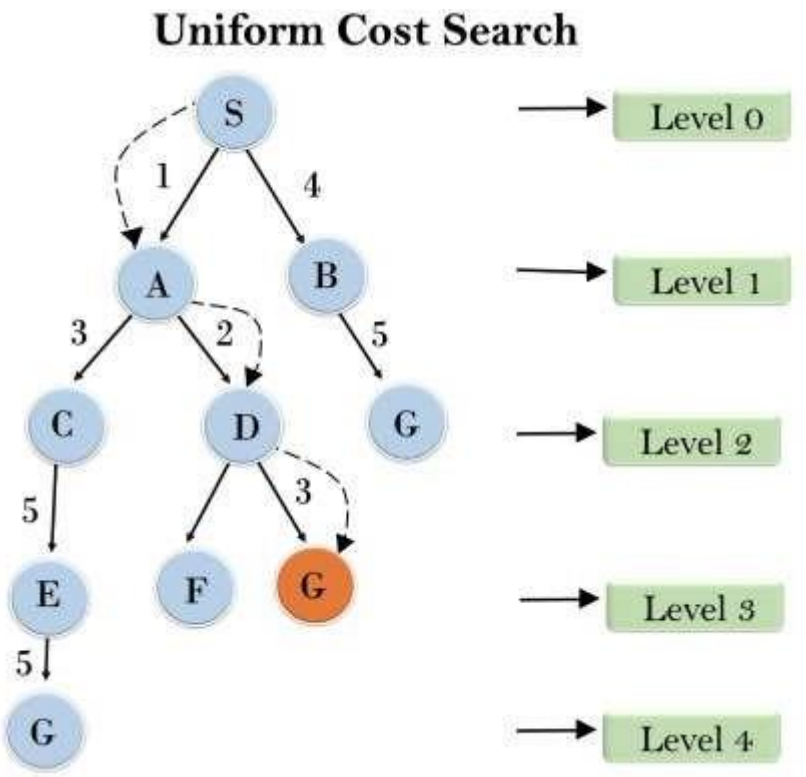
Advantages:

- Uniform cost search is optimal because at every state the path with the least cost is chosen.

Disadvantages:

It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

Example:



Completeness:

Uniform-cost search is complete, such as if there is a solution, UCS will find it.

Time Complexity:

Let C* is Cost of the optimal solution, and ε is each step to get closer to the goal node. Then the number of steps is = C*/ε+1. Here we have taken +1, as we start from state 0 and end to C*/ε.

Hence, the worst-case time complexity of Uniform-cost search is O(b1 + [C*/ε])/.

Space Complexity:

The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is O(b1 + [C*/ε]).

Optimal:

Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

5. Iterative deepening depth-first Search:

The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.

This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.

This Search algorithm combines the benefits of Breadth-first search's fast search and depth- first search's memory efficiency.

The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

Advantages:

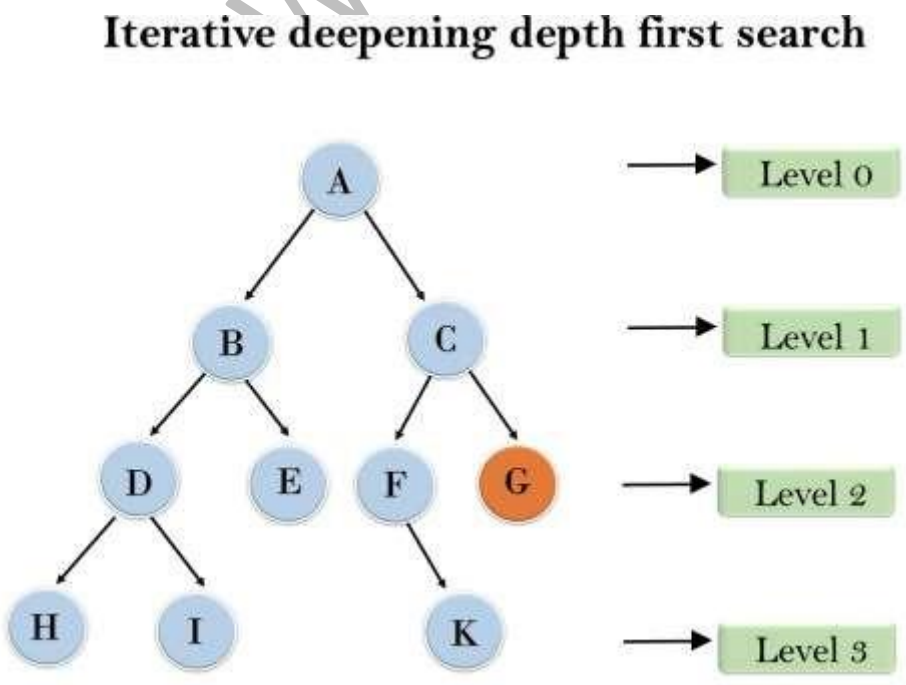
- It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

Disadvantages:

- The main drawback of IDDFS is that it repeats all the work of the previous phase.

Example:

Following tree structure is showing the iterative deepening depth-first search. IDDFS algorithm performs various iterations until it does not find the goal node. The iteration performed by the algorithm is given as:



1'st Iteration > A

2'nd Iteration > A, B, C

3'rd Iteration >A, B, D, E, C, F, G

In the third iteration, the algorithm will find the goal node.

Completeness:

This algorithm is complete is if the branching factor is finite.

Time Complexity:

Let's suppose b is the branching factor and depth is d then the worst-case time complexity is O(bd).

Space Complexity:

The space complexity of IDDFS will be O(bd).

Optimal:

IDDFS algorithm is optimal if path cost is a non- decreasing function of the depth of the node.

6. Bidirectional Search Algorithm:

Bidirectional search algorithm runs two simultaneous searches, one form initial state called as forward-search and other from goal node called as backward-search, to find the goal node. Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex. The search stops when these two graphs intersect each other.

Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.

Advantages:

- Bidirectional search is fast.
- Bidirectional search requires less memory

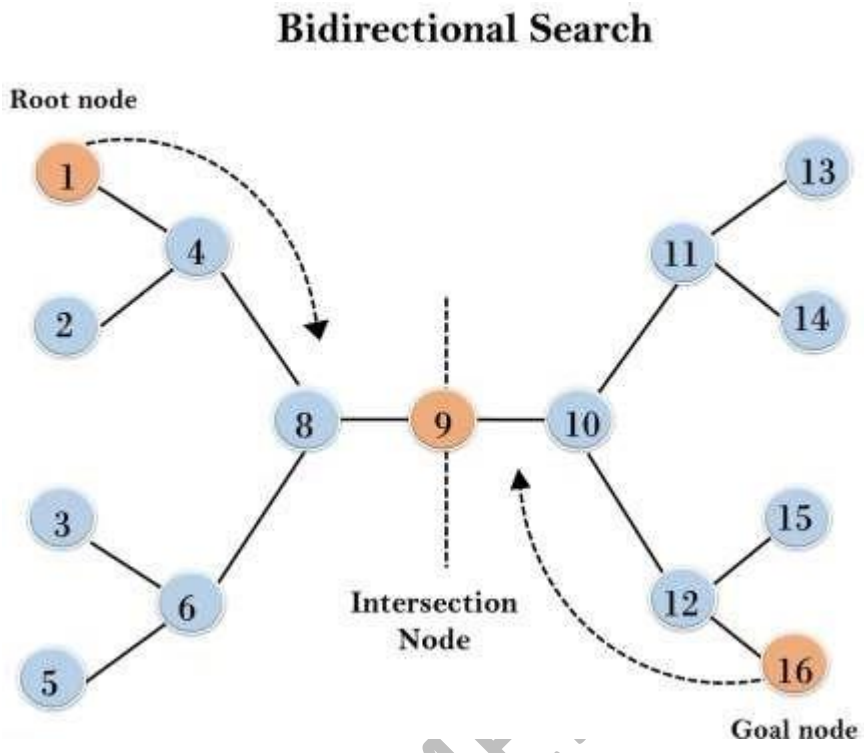
Disadvantages:

- Implementation of the bidirectional search tree is difficult.
- In bidirectional search, one should know the goal state in advance.

Example:

In the below search tree, bidirectional search algorithm is applied. This algorithm divides one graph/tree into two sub-graphs. It starts traversing from node 1 in the forward direction and starts from goal node 16 in the backward direction.

The algorithm terminates at node 9 where two searches meet.



Completeness: Bidirectional Search is complete if we use BFS in both searches.

Time Complexity: Time complexity of bidirectional search using BFS is O(bd).

Space Complexity: Space complexity of bidirectional search is O(bd).

Optimal: Bidirectional search is Optimal.

Informed Search Algorithms

Informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach to goal node, etc. This knowledge help agents to explore less to the search space and find more efficiently the goal node.

The informed search algorithm is more useful for large search space. Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.

Heuristics function: Heuristic is a function which is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal.

The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time.

Heuristic function estimates how close a state is to the goal.

It is represented by h(n), and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.

Admissibility of the heuristic function is given as:

h(n) <= h*(n)

Here h(n) is heuristic cost, and h*(n) is the estimated cost. Hence heuristic cost should be less than or equal to the estimated cost.

Pure Heuristic Search:

Pure heuristic search is the simplest form of heuristic search algorithms. It expands nodes based on their heuristic value h(n). It maintains two lists, OPEN and CLOSED list. In the CLOSED list, it places those nodes which have already expanded and in the OPEN list, it places nodes which have yet not been expanded.

On each iteration, each node n with the lowest heuristic value is expanded and generates all its successors and n is placed to the closed list. The algorithm continues unit a goal state is found.

In the informed search we will discuss two main algorithms which are given below:

- Best First Search Algorithm(Greedy search)
- A* Search Algorithm

1. Best-first Search Algorithm (Greedy Search):

Greedy best-first search algorithm always selects the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms.

It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms. With the help of best-first search, at each step, we can choose the most promising node. In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e.

f(n)= g(n).

Were, h(n)= estimated cost from node n to the goal.

The greedy best first algorithm is implemented by the priority queue.

Best first search algorithm:

- Step 1: Place the starting node into the OPEN list.
- Step 2: If the OPEN list is empty, Stop and return failure.
- Step 3: Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.

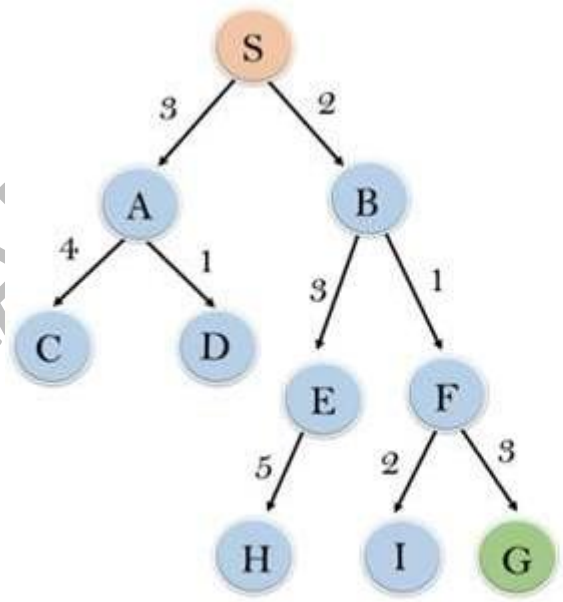
- Step 4: Expand the node n, and generate the successors of node n.
- Step 5: Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
- Step 6: For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.
- Step 7: Return to Step 2.

Advantages:

- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
- This algorithm is more efficient than BFS and DFS algorithms.
- **Disadvantages:**
- It can behave as an unguided depth-first search in the worst case scenario.
- It can get stuck in a loop as DFS.
- This algorithm is not optimal.

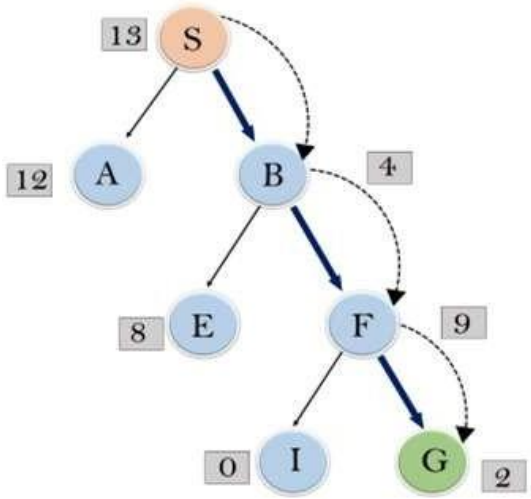
Example:

Consider the below search problem, and we will traverse it using greedy best-first search. At each iteration, each node is expanded using evaluation function f(n)=h(n) , which is given in the below table.



node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

In this search example, we are using two lists which are OPEN and CLOSED Lists. Following are the iteration for traversing the above example.



Expand the nodes of S and put in the CLOSED list

Initialization: Open [A, B], Closed [S]

Iteration 1: Open [A], Closed [S, B]

Iteration 2: Open [E, F, A], Closed [S, B]

: Open [E, A], Closed [S, B, F]

Iteration 3: Open [I, G, E, A], Closed [S, B, F]

: Open [I, E, A], Closed [S, B, F, G]

Hence the final solution path will be: S----> B----->F > G

Time Complexity: The worst case time complexity of Greedy best first search is O(bm).

Space Complexity: The worst case space complexity of Greedy best first search is O(bm). Where, m is the maximum depth of the search space.

Complete: Greedy best-first search is also incomplete, even if the given state space is finite.

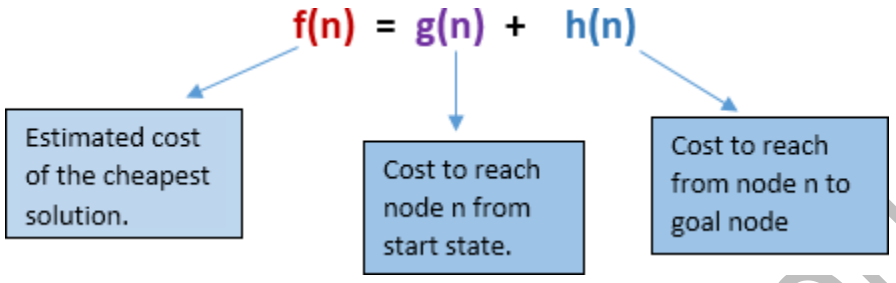
Optimal: Greedy best first search algorithm is not optimal.

2. A* Search Algorithm:

A* search is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n). It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently. A* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less

search tree and provides optimal result faster. A* algorithm is similar to UCS except that it uses g(n)+h(n) instead of g(n).

In A* search algorithm, we use search heuristic as well as the cost to reach the node. Hence we can combine both costs as following, and this sum is called as a fitness number.



At each point in the search space, only those node is expanded which have the lowest value of f(n), and the algorithm terminates when the goal node is found.

Algorithm of A* search:

Step1: Place the starting node in the OPEN list.

Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.

Step 3: Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise

Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.

Step 6: Return to Step 2.

Advantages:

- A* search algorithm is the best algorithm than other search algorithms.
- A* search algorithm is optimal and complete.
- This algorithm can solve very complex problems.

Disadvantages:

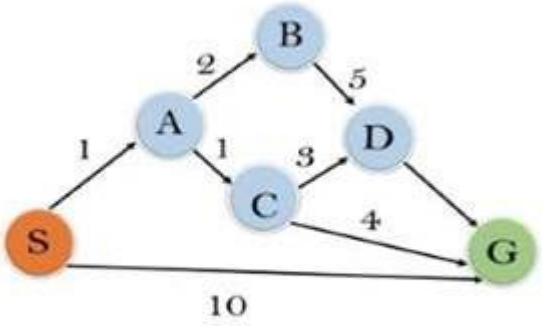
- It does not always produce the shortest path as it mostly based on heuristics and approximation.

- A* search algorithm has some complexity issues.
- The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

Example:

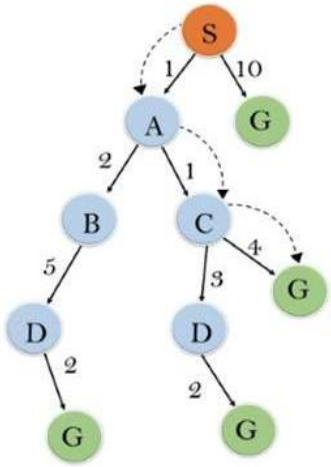
In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state.

Here we will use OPEN and CLOSED list.



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

Solution:



Initialization: {(S, 5)}

Iteration1: {(S--> A, 4), (S-->G, 10)}

Iteration2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}

Iteration3: {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}

Iteration 4 will give the final result, as S--->A--->C--->G it provides the optimal path with cost 6.

Points to remember:

- A* algorithm returns the path which occurred first, and it does not search for all remaining paths.
- The efficiency of A* algorithm depends on the quality of heuristic.
- A* algorithm expands all nodes which satisfy the condition f(n)<="" li="">

Complete: A* algorithm is complete as long as:

- Branching factor is finite.
- Cost at every action is fixed.

Optimal: A* search algorithm is optimal if it follows below two conditions:

- **Admissible:** the first condition requires for optimality is that h(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.
- **Consistency:** Second required condition is consistency for only A* graph-search.

If the heuristic function is admissible, then A* tree search will always find the least cost path.

Time Complexity: The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time complexity is O(b^d), where b is the branching factor.

Space Complexity: The space complexity of A* search algorithm is O(b^d)

Local Search Algorithms

Hill Climbing Algorithm in Artificial Intelligence

- 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:

Following are some main features of Hill Climbing Algorithm:

- **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.

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:

- Less time consuming
- Less optimal solution and the solution is not guaranteed

Algorithm for Simple Hill Climbing:

- Step 1: Evaluate the initial state, if it is goal state then return success and Stop.
- Step 2: Loop Until a solution is found or there is no new operator left to apply.
- Step 3: Select and apply an operator to the current state.
- Step 4: Check new state:
    1. If it is goal state, then return success and quit.

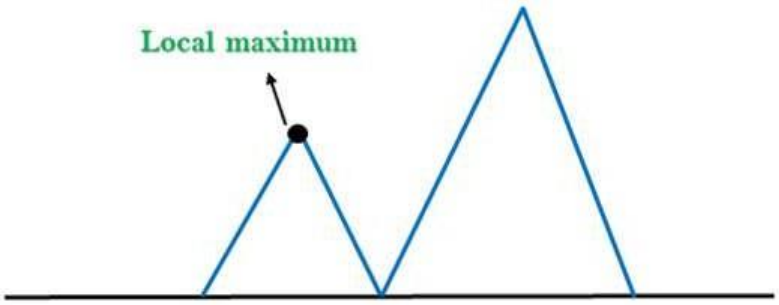
2. Else if it is better than the current state then assign new state as a current state.
3. Else if not better than the current state, then return to step2.

- Step 5: Exit.

Problems in Hill Climbing Algorithm:

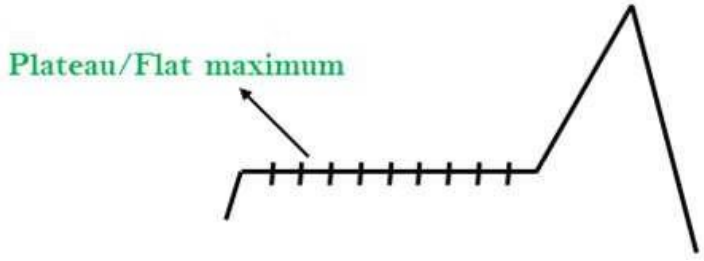
1. Local Maximum: A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.

Solution: Backtracking technique can be a solution of the local maximum in state space landscape. Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well.



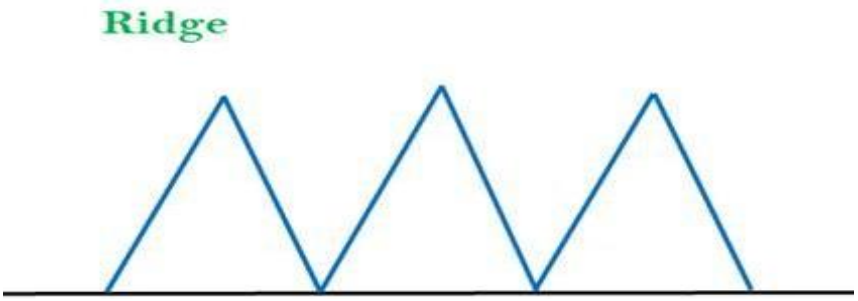
2. Plateau: A plateau is the flat area of the search space in which all the neighbor states of the current state contains the same value, because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area.

Solution: The solution for the plateau is to take big steps or very little steps while searching, to solve the problem. Randomly select a state which is far away from the current state so it is possible that the algorithm could find non-plateau region.



3. Ridges: A ridge is a special form of the local maximum. It has an area which is higher than its surrounding areas, but itself has a slope, and cannot be reached in a single move.

Solution: With the use of bidirectional search, or by moving in different directions, we can improve this problem.



Solution:

Simulated Annealing:

A hill-climbing algorithm which never makes a move towards a lower value guaranteed to be incomplete because it can get stuck on a local maximum. And if algorithm applies a random walk, by moving a successor, then it may complete but not efficient. Simulated Annealing is an algorithm which yields both efficiency and completeness.

Simulated annealing in which the algorithm picks a random move, instead of picking the best move. If the random move improves the state, then it follows the same path. Otherwise, the algorithm follows the path which has a probability of less than 1 or it moves downhill and chooses another path.

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.

Travelling Salesman Problem

In this algorithm, the objective is to find a low-cost tour that starts from a city, visits all cities en-route exactly once and ends at the same starting city.

Means-Ends Analysis in Artificial Intelligence

- We have studied the strategies which can reason either in forward or backward, but a mixture of the two directions is appropriate for solving a complex and large problem. Such a mixed strategy, make it possible that first to solve the major part of a problem and then

go back and solve the small problems arise during combining the big parts of the problem. Such a technique is called Means-Ends Analysis.

- Means-Ends Analysis is problem-solving techniques used in Artificial intelligence for limiting search in AI programs.
- It is a mixture of Backward and forward search technique.
- The MEA technique was first introduced in 1961 by Allen Newell, and Herbert A. Simon in their problem-solving computer program, which was named as General Problem Solver (GPS).
- The MEA analysis process centered on the evaluation of the difference between the current state and goal state.

How means-ends analysis Works:

The means-ends analysis process can be applied recursively for a problem. It is a strategy to control search in problem-solving. Following are the main Steps which describes the working of MEA technique for solving a problem.

a. First, evaluate the difference between Initial State and final State.
b. Select the various operators which can be applied for each difference.
c. Apply the operator at each difference, which reduces the difference between the current state and goal state.

Operator Subgoaling

In the MEA process, we detect the differences between the current state and goal state. Once these differences occur, then we can apply an operator to reduce the differences. But sometimes it is possible that an operator cannot be applied to the current state. So we create the subproblem of the current state, in which operator can be applied, such type of backward chaining in which operators are selected, and then sub goals are set up to establish the preconditions of the operator is called Operator Subgoaling.

Algorithm for Means-Ends Analysis:

Let's we take Current state as CURRENT and Goal State as GOAL, then following are the steps for the MEA algorithm.

- Step 1: Compare CURRENT to GOAL, if there are no differences between both then return Success and Exit.
- Step 2: Else, select the most significant difference and reduce it by doing the following steps until the success or failure occurs.

1. Select a new operator O which is applicable for the current difference, and if there is no such operator, then signal failure.
2. Attempt to apply operator O to CURRENT. Make a description of two states.
    i. O-Start, a state in which O's preconditions are satisfied.
    ii. O-Result, the state that would result if O were applied In O-start.
3. If

    (First-Part< MEA(CURRENT,O-START)

    And

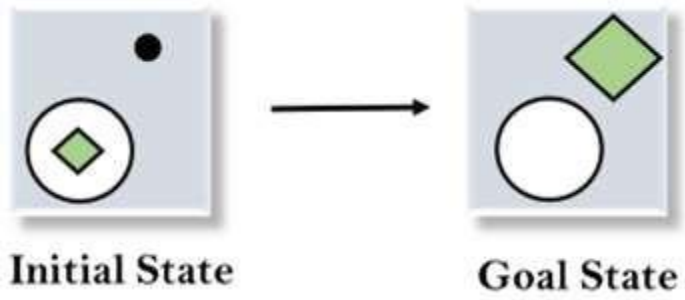
    (LAST-Part <----- MEA (O-Result, GOAL), are successful, then signal Success and return the result of combining FIRST-PART, O, and LAST-PART.

The above-discussed algorithm is more suitable for a simple problem and not adequate for solving complex problems.

C++ vs Java

Example of Mean-Ends Analysis:

Let's take an example where we know the initial state and goal state as given below. In this problem, we need to get the goal state by finding differences between the initial state and goal state and applying operators.



Solution:

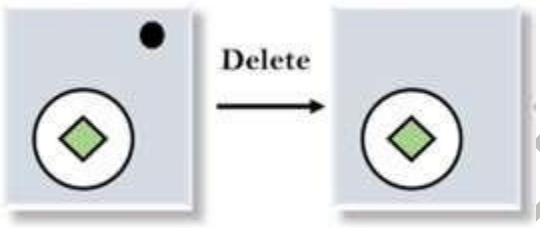
To solve the above problem, we will first find the differences between initial states and goal states, and for each difference, we will generate a new state and will apply the operators. The operators we have for this problem are:

- Move
- Delete
- Expand

1. **Evaluating the initial state:** In the first step, we will evaluate the initial state and will compare the initial and Goal state to find the differences between both states.

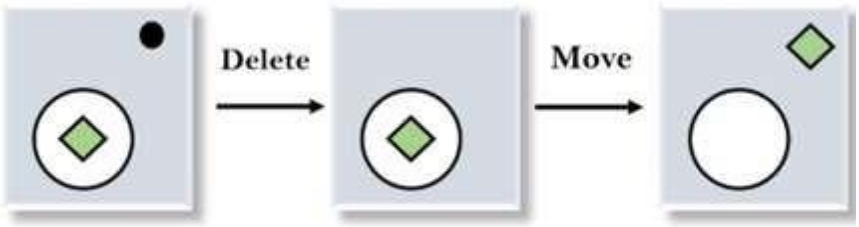
Initial state

2. Applying Delete operator: As we can check the first difference is that in goal state there is no dot symbol which is present in the initial state, so, first we will apply the Delete operator to remove this dot.



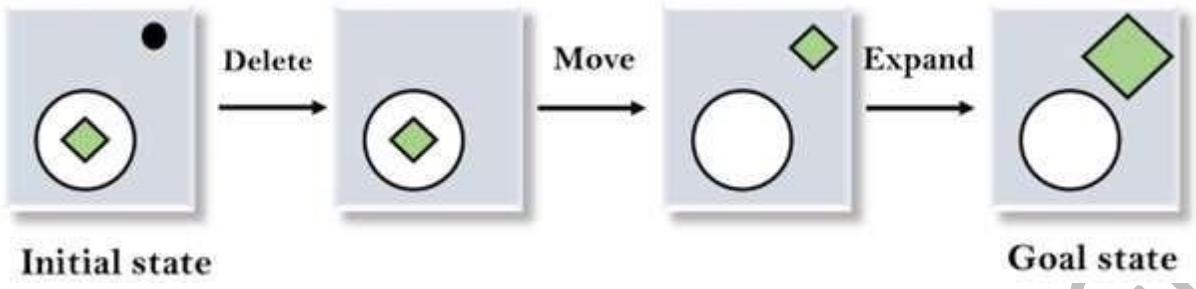
Initial state

3. Applying Move Operator: After applying the Delete operator, the new state occurs which we will again compare with goal state. After comparing these states, there is another difference that is the square is outside the circle, so, we will apply the Move Operator.



Initial state

4. Applying Expand Operator: Now a new state is generated in the third step, and we will compare this state with the goal state. After comparing the states there is still one difference which is the size of the square, so, we will apply Expand operator, and finally, it will generate the goal state.



Adversarial Search

Adversarial search is a search, where we examine the problem which arises when we try to plan ahead of the world and other agents are planning against us.

- In previous topics, we have studied the search strategies which are only associated with a single agent that aims to find the solution which often expressed in the form of a sequence of actions.
- But, there might be some situations where more than one agent is searching for the solution in the same search space, and this situation usually occurs in game playing.
- The environment with more than one agent is termed as multi-agent environment, in which each agent is an opponent of other agent and playing against each other. Each agent needs to consider the action of other agent and effect of that action on their performance.
- So, Searches in which two or more players with conflicting goals are trying to explore the same search space for the solution, are called adversarial searches, often known as Games.
- Games are modeled as a Search problem and heuristic evaluation function, and these are the two main factors which help to model and solve games in AI.

Types of Games in AI:

	Deterministic	**Chance Moves**
Perfect information	Chess, Checkers, go, Othello	Backgammon, monopoly
Imperfect information	Battleships, blind, tic-tac-toe	Bridge, poker, scrabble, nuclear war

- **Perfect information:** A game with the perfect information is that in which agents can look into the complete board. Agents have all the information about the game, and they can see each other moves also. Examples are Chess, Checkers, Go, etc.
- **Imperfect information:** If in a game agents do not have all information about the game and not aware with what's going on, such type of games are called the game with imperfect information, such as tic-tac-toe, Battleship, blind, Bridge, etc.
- **Deterministic games:** Deterministic games are those games which follow a strict pattern and set of rules for the games, and there is no randomness associated with them. Examples are chess, Checkers, Go, tic-tac-toe, etc.
- **Non-deterministic games:** Non-deterministic are those games which have various unpredictable events and has a factor of chance or luck. This factor of chance or luck is introduced by either dice or cards. These are random, and each action response is not fixed. Such games are also called as stochastic games. Example: Backgammon, Monopoly, Poker, etc.

Zero-Sum Game

- Zero-sum games are adversarial search which involves pure competition.
- In Zero-sum game each agent's gain or loss of utility is exactly balanced by the losses or gains of utility of another agent.
- One player of the game try to maximize one single value, while other player tries to minimize it.
- Each move by one player in the game is called as ply.
- Chess and tic-tac-toe are examples of a Zero-sum game.

Zero-sum game: Embedded thinking

The Zero-sum game involved embedded thinking in which one agent or player is trying to figure out:

- What to do.
- How to decide the move
- Needs to think about his opponent as well
- The opponent also thinks what to do

Each of the players is trying to find out the response of his opponent to their actions. This requires embedded thinking or backward reasoning to solve the game problems in AI.

Formalization of the problem:

A game can be defined as a type of search in AI which can be formalized of the following elements:

- **Initial state:** It specifies how the game is set up at the start.
- **Player(s):** It specifies which player has moved in the state space.
- **Action(s):** It returns the set of legal moves in state space.
- **Result(s, a):** It is the transition model, which specifies the result of moves in the state space.
- **Terminal-Test(s):** Terminal test is true if the game is over, else it is false at any case. The state where the game ends is called terminal states.
- **Utility(s, p):** A utility function gives the final numeric value for a game that ends in terminal states s for player p. It is also called payoff function. For Chess, the outcomes are a win, loss, or draw and its payoff values are +1, 0, ½. And for tic-tac-toe, utility values are +1, -1, and 0.

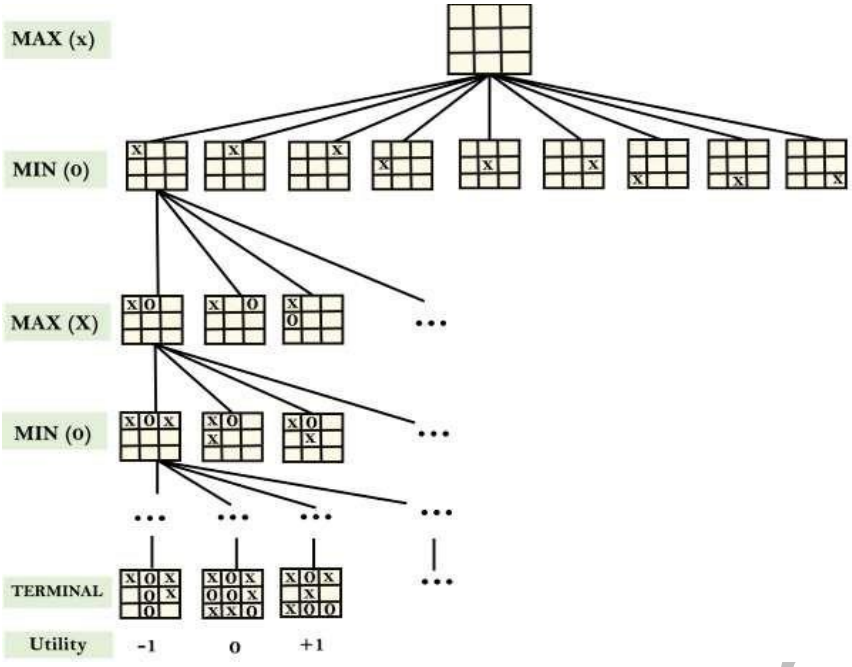
Game tree:

A game tree is a tree where nodes of the tree are the game states and Edges of the tree are the moves by players. Game tree involves initial state, actions function, and result Function.

Example: Tic-Tac-Toe game tree:

The following figure is showing part of the game-tree for tic-tac-toe game. Following are some key points of the game:

- There are two players MAX and MIN.
- Players have an alternate turn and start with MAX.
- MAX maximizes the result of the game tree
- MIN minimizes the result.



Example Explanation:

- From the initial state, MAX has 9 possible moves as he starts first. MAX place x and MIN place o, and both player plays alternatively until we reach a leaf node where one player has three in a row or all squares are filled.
- Both players will compute each node, minimax, the minimax value which is the best achievable utility against an optimal adversary.
- Suppose both the players are well aware of the tic-tac-toe and playing the best play. Each player is doing his best to prevent another one from winning. MIN is acting against Max in the game.
- So in the game tree, we have a layer of Max, a layer of MIN, and each layer is called as Ply. Max place x, then MIN puts o to prevent Max from winning, and this game continues until the terminal node.
- In this either MIN wins, MAX wins, or it's a draw. This game-tree is the whole search space of possibilities that MIN and MAX are playing tic-tac-toe and taking turns alternately.

Adversarial search is a game-playing technique where the agents are surrounded by a competitive environment. A conflicting goal is given to the agents (multi agent). These agents compete with one another and try to defeat one another in order to win the game. Such conflicting goals give rise to the adversarial search. Here, game-playing means discussing those games where human intelligence and logic factor is used, excluding other factors such as luck factor. Tic-tac-toe, chess, checkers, etc., are such type of games where no luck factor works, only mind works.

Mathematically, this search is based on the concept of 'Game Theory.' According to game theory, a game is played between two players. To complete the game, one has to win the game and the other looses automatically.

Techniques required to get the best optimal solution

There is always a need to choose those algorithms which provide the best optimal solution in a limited time. So, we use the following techniques which could fulfill our requirements:

- **Pruning:** A technique which allows ignoring the unwanted portions of a search tree which make no difference in its final result.
- **Heuristic Evaluation Function:** It allows to approximate the cost value at each levelof the search tree, before reaching the goal node.

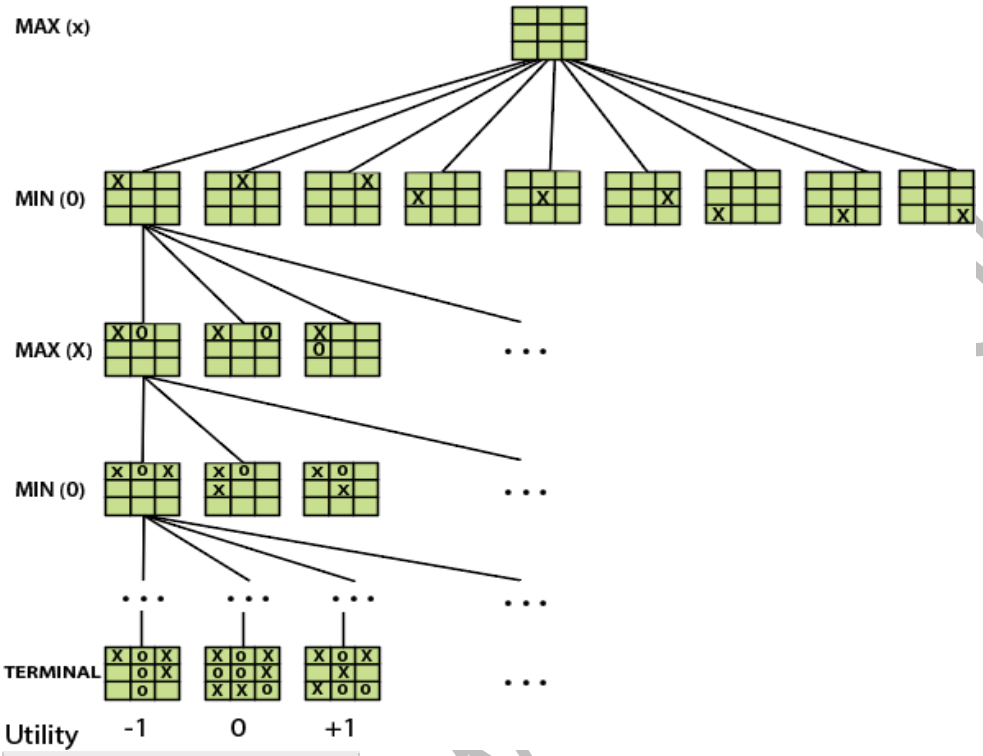
Elements of Game Playing search

To play a game, we use a game tree to know all the possible choices and to pick the best one out. There are following elements of a game-playing:

- **S0:** It is the initial state from where a game begins.
- **PLAYER (s):** It defines which player is having the current turn to make a move in the state.
- **ACTIONS (s):** It defines the set of legal moves to be used in a state.
- **RESULT (s, a):** It is a transition model which defines the result of a move.
- **TERMINAL-TEST (s):** It defines that the game has ended and returns true.
- **UTILITY (s,p):** It defines the final value with which the game has ended. This function is also known as Objective function or Payoff function. The price which the winner will get i.e.
- **(-1):** If the PLAYER loses.
- **(+1):** If the PLAYER wins.
- **(0):** If there is a draw between the PLAYERS.

For example, in chess, tic-tac-toe, we have two or three possible outcomes. Either to win, to lose, or to draw the match with values +1,-1 or 0.

Let's understand the working of the elements with the help of a game tree designed for tic-tac-toe. Here, the node represents the game state and edges represent the moves taken by the players.



A game-tree for tic-tac-

- **INITIAL STATE (S0):** The top node in the game-tree represents the initial state in the tree and shows all the possible choice to pick out one.
- **PLAYER (s):** There are two players, MAX and MIN. MAX begins the game by picking one best move and place X in the empty square box.
- **ACTIONS (s):** Both the players can make moves in the empty boxes chance by chance.
- **RESULT (s, a):** The moves made by MIN and MAX will decide the outcome of the game.
- **TERMINAL-TEST(s):** When all the empty boxes will be filled, it will be the terminating state of the game.
- **UTILITY:** At the end, we will get to know who wins: MAX or MIN, and accordingly, the price will be given to them.

Types of Games in AI:

	Deterministic	Chance Moves
Perfect information	Chess, Checkers, go, Othello	Backgammon, monopoly
Imperfect information	Battleships, blind, tic-tac-toe	Bridge, poker, scrabble, nuclear war

- Perfect information: A game with the perfect information is that in which agents can look into the complete board. Agents have all the information about the game, and they can see each other moves also. Examples are Chess, Checkers, Go, etc.
- Imperfect information: If in a game agents do not have all information about the game and not aware with what's going on, such type of games are called the game with imperfect information, such as tic-tac-toe, Battleship, blind, Bridge, etc.
- Deterministic games: Deterministic games are those games which follow a strict pattern and set of rules for the games, and there is no randomness associated with them. Examples are chess, Checkers, Go, tic-tac-toe, etc.
- Non-deterministic games: Non-deterministic are those games which have various unpredictable events and has a factor of chance or luck. This factor of chance or luck is introduced by either dice or cards. These are random, and each action response is not fixed. Such games are also called as stochastic games. Example: Backgammon, Monopoly, Poker, etc.

Zero-Sum Game

- Zero-sum games are adversarial search which involves pure competition.
- In Zero-sum game each agent's gain or loss of utility is exactly balanced by the losses or gains of utility of another agent.
- One player of the game try to maximize one single value, while other player tries to minimize it.
- Each move by one player in the game is called as ply.
- Chess and tic-tac-toe are examples of a Zero-sum game.

Zero-sum game: Embedded thinking

The Zero-sum game involved embedded thinking in which one agent or player is trying to figure out:

- What to do.
- How to decide the move
- Needs to think about his opponent as well
- The opponent also thinks what to do

Each of the players is trying to find out the response of his opponent to their actions. This requires embedded thinking or backward reasoning to solve the game problems in AI.

Types of algorithms in Adversarial search

In a normal search, we follow a sequence of actions to reach the goal or to finish the game optimally. But in an adversarial search, the result depends on the players which will decide the result of the game. It is also obvious that the solution for the goal state will be an optimal solution because the player will try to win the game with the shortest path and under limited time.

There are following types of adversarial search:

- Minmax Algorithm
- Alpha-beta Pruning.

Minimax

In artificial intelligence, minimax is a decision-making strategy under game theory, which is used to minimize the losing chances in a game and to maximize the winning chances. This strategy is also known as 'Minmax,' 'MM,' or 'Saddle point.' Basically, it is a two-player game strategy where if one wins, the other loose the game. This strategy simulates those games that we play in our day-to-day life. Like, if two persons are playing chess, the result will be in favor of one player and will unfavor the other one. The person who will make his best try,efforts as well as cleverness, will surely win.

We can easily understand this strategy via game tree– where the nodes represent the states of the game and edges represent the moves made by the players in the game. Players will be two namely:

- **MIN:** Decrease the chances of MAX to win the game.
- **MAX:** Increases his chances of winning the game.

They both play the game alternatively, i.e., turn by turn and following the above strategy, i.e., if one wins, the other will definitely lose it. Both players look at one another as competitors and will try to defeat one-another, giving their best.

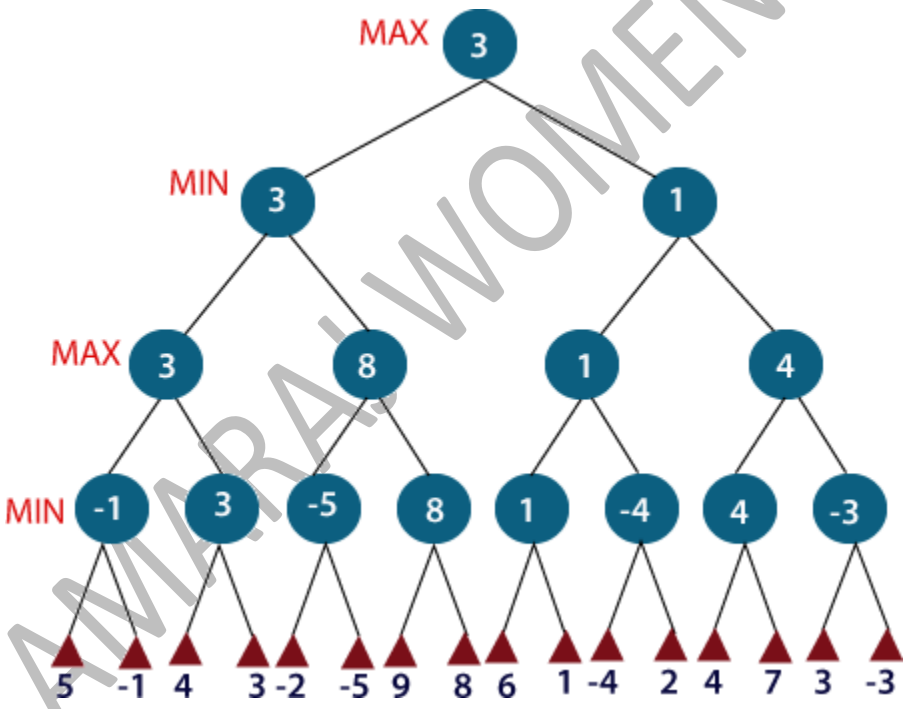
In minimax strategy, the result of the game or the utility value is generated by a heuristic function by propagating from the initial node to the root node. It follows the backtracking technique and

backtracks to find the best choice. MAX will choose that path which will increase its utility value and MIN will choose the opposite path which could help it to minimize MAX's utility value.

MINIMAX Algorithm

MINIMAX algorithm is a backtracking algorithm where it backtracks to pick the best move out of several choices. MINIMAX strategy follows the DFS (Depth-first search) concept. Here, we have two players MIN and MAX, and the game is played alternatively between them, i.e., when MAX made a move, then the next turn is of MIN. It means the move made by MAX is fixed and, he cannot change it. The same concept is followed in DFS strategy, i.e., we follow the same path and cannot change in the middle. That's why in MINIMAX algorithm, instead of BFS, we follow DFS.

- Keep on generating the game tree/ search tree till a limit d.
- Compute the move using a heuristic function.
- Propagate the values from the leaf node till the current position following the minimax strategy.
- Make the best move from the choices.



For example, in the above figure, the two players MAX and MIN are there. MAX starts the game by choosing one path and propagating all the nodes of that path. Now, MAX will backtrack to the initial node and choose the best path where his utility value will be the maximum. After this, its MIN chance. MIN will also propagate through a path and again will backtrack, but MIN will choose the path which could minimize MAX winning chances or the utility value.

So, if the level is minimizing, the node will accept the minimum value from the successor nodes. If the level is maximizing, the node will accept the maximum value from the successor.

Alpha-beta pruning:

Alpha-beta pruning is an advance version of MINIMAX algorithm. The drawback of minimax strategy is that it explores each node in the tree deeply to provide the best path among all the paths. This increases its time complexity. But as we know, the performance measure is the first consideration for any optimal algorithm. Therefore, alpha-beta pruning reduces this drawback of minimax strategy by less exploring the nodes of the search tree.

The method used in alpha-beta pruning is that it cutoff the search by exploring less number of nodes. It makes the same moves as a minimax algorithm does, but it prunes the unwanted branches using the pruning technique (discussed in adversarial search). Alpha-beta pruning works on two threshold values, i.e., alpha and beta

- It is the best highest value, a MAX player can have. It is the lower bound, which represents negative infinity value.
- It is the best lowest value, a MIN player can have. It is the upper bound which represents positive infinity.

So, each MAX node has value, which never decreases, and each MIN node has value, which never increases.

Note: Alpha-beta pruning technique can be applied to trees of any depth, and it is possible to prune the entire sub trees easily.

UNIT - III

Probabilistic reasoning:

Probabilistic reasoning is a way of knowledge representation where we apply the concept of probability to indicate the uncertainty in knowledge.

AI system uses probabilistic reasoning to model and reason about uncertain situations (i.e.,it provides a way to handle the uncertainty).

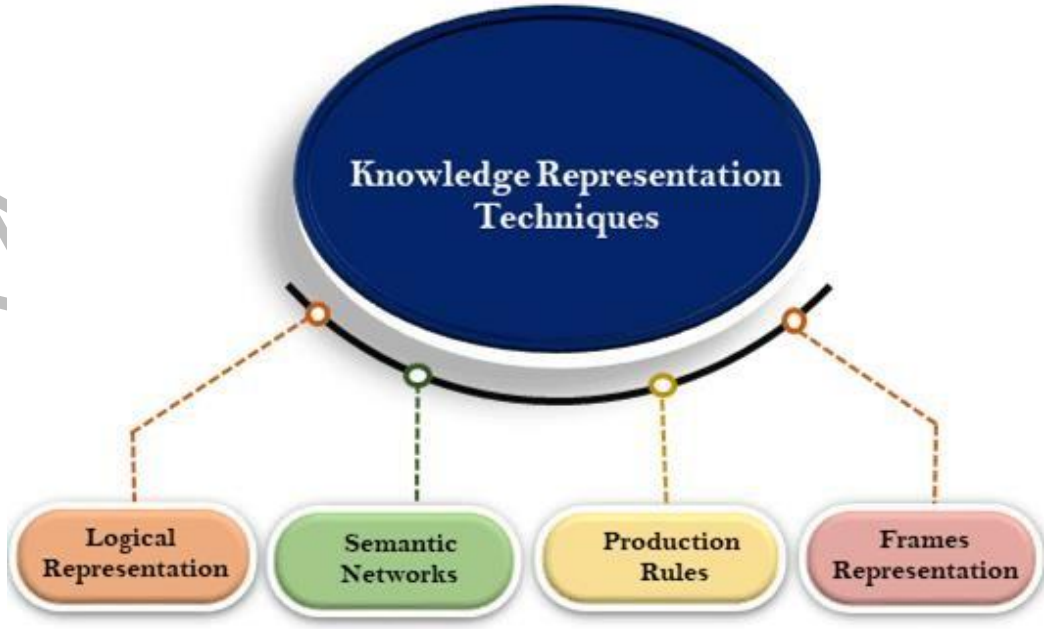
In the real world, there are lots of scenarios, where the certainty of something is not confirmed, such as "It will rain today," "behavior of someone for some situations," "A match between two teams or two players." These are probable sentences for which we can assume that it will happen but not sure about it, so here we use probabilistic reasoning.

Knowledge Representation:

It is a way which describes how we can represent knowledge in artificial intelligence. It is a fundamental concept in AI that involves creating models and structures to represent information and knowledge in a way that intelligent systems can use.

Knowledge representation is not just storing data into some database, but it also enables an intelligent machine to learn from that knowledge and experiences so that it can behave intelligently like a human. It is responsible for representing information about the real world so that a computer can understand and can utilize this knowledge to solve the complex real world problems such as diagnosis a medical condition or communicating with humans in natural language.

Techniques of knowledge representation



- **Logical representation:** This involves representing knowledge in a symbolic logic or rule-based system, which uses formal languages to express and infer new knowledge.
- **Semantic networks:** This involves representing knowledge through nodes and links, where nodes represent concepts or objects, and links represent their relationships.
- **Production rules:** In an AI production system, rules encode knowledge and specify how the system should respond to different inputs and conditions. Production rules consist of conditions (if part) and actions (then part), which are applied based on the system's current state and available data.
- **Frames:** This approach involves representing knowledge in the form of structures called frames, which capture the properties and attributes of objects or concepts and the relationships between them.
- **Neural networks:** This involves representing knowledge in the form of patterns or connections between nodes in a network, which can be used to learn and infer new knowledge from data.

Uncertainty:

In AI, knowledge representation uses techniques such as first-order logic and propositional logic with certainty, which means we were sure about predicates (factor or relation which connects two objects).

With this knowledge representation, suppose A and B are two statements, if we implement if-then rule to these statements, we might write A→B, which means if A is true then B is true, or if A is false then B is false, if A is true then B is false, if A is false then B is true.

But consider a situation where we are not sure about whether A is true or not then we cannot express this statement, this situation is called uncertainty. It refers to situations where there is a lack of complete information or knowledge about a particular aspect, leading to ambiguity and unpredictability.

So to represent uncertain knowledge, where we are not sure about the predicates, we need uncertain reasoning or probabilistic reasoning.

Causes of uncertainty:

Following are some leading causes of uncertainty to occur in the real world.

1. Information occurred from unreliable sources.
2. Experimental Errors
3. Equipment fault
4. Temperature variation
5. Climate change.

Need of probabilistic reasoning in AI:

- When there are unpredictable outcomes.
- When specifications or possibilities of predicates becomes too large to handle.
- When an unknown error occurs during an experiment.

As probabilistic reasoning uses probability and related terms, so before understanding probabilistic reasoning let's understand Probability:

Probability:

Probability can be defined as a chance that an uncertain event will occur. It is the numerical measure of the likelihood that an event will occur. The value of probability always remains between 0 and 1 that represent ideal uncertainties.

$0 \leq P(A) \leq 1$, where P(A) is the probability of an event A.

P(A) = 0, indicates total uncertainty in an event A.

P(A) =1, indicates total certainty in an event A.

P(¬A) = probability of a not happening event.

P(¬A) + P(A) = 1.

We can find the probability of an uncertain event by using the below formula.

$$\text{Probability of occurrence} = \frac{\text{Number of desired outcomes}}{\text{Total number of outcomes}}$$

More Terms:

Event: Each possible outcome of a variable is called an event.

Sample space: The collection of all possible events is called sample space.

Random variables: Random variables are used to represent the events and objects in the real world.

Prior probability: The prior probability of an event is probability computed before observing new information.

Posterior Probability: The probability that is calculated after all evidence or information has taken into account. It is a combination of prior probability and new information.

Prior probability:

Degree of belief in an event, in the absence of any other information

Example:

- P(rain tomorrow) = 0.7
- P(no rain tomorrow) = 0.3

Conditional probability:

Conditional probability is a probability of occurring an event when another event has already happened.

What is the probability of an event, given knowledge of another event.

Example:

- P(raining | sunny)
- P(raining | cloudy)
- P(raining | cloudy, cold)

Let's suppose, we want to calculate the event A when event B has already occurred, "the probability of A under the conditions of B", it can be written as:

$$P(A|B) = \frac{P(A \wedge B)}{P(B)}$$

Where P(A⋀B)= Joint probability of a and B

P(B)= Marginal probability of B.

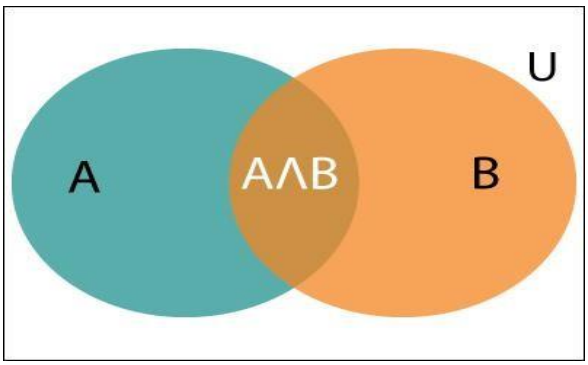
If the probability of A is given and we need to find the probability of B, then it will be given as:

$$P(B|A) = \frac{P(A \wedge B)}{P(A)}$$

Advantage is that in some cases, given knowledge of one or more random variables, we can improve our prior belief of another random variable.

Example:

- P(slept in stadium) = 0.5
- P(slept in stadium | liked match) = 0.33
- P(didn’t slept in stadium | liked match) = 0.67



It can be explained by using the below Venn diagram, where B is occurred event, so sample space will be reduced to set B, and now we can only calculate event A when event B is already occurred by dividing the probability of P(A∧B) by P (B).

Example:

In a class, there are 70% of the students who like English and 40% of the students who likes English and mathematics, and then what is the percent of students those who like English also like mathematics?

Solution:

Let, A is an event that a student likes Mathematics

B is an event that a student likes English.

$$P(A|B) = \frac{P(A \wedge B)}{P(B)} = \frac{0.4}{0.7} = 57\%$$

Hence, 57% are the students who like English also like Mathematics.

In probabilistic reasoning, there are two ways to solve problems with uncertain knowledge:

- Bayes' rule or Bayes' Theorem
- Bayesian Statistics

Bayes' theorem:

Bayes' theorem is also known as Bayes' rule, Bayes' law, or Bayesian reasoning, which determines the probability of an event with uncertain knowledge.

In probability theory, it relates the conditional probability and marginal probabilities of two random events.

Bayes' theorem was named after the British mathematician Thomas Bayes. The Bayesian inference is an application of Bayes' theorem, which is fundamental to Bayesian statistics.

It is a way to calculate the value of P(B|A) with the knowledge of P(A|B).

Bayes' theorem allows updating the probability prediction of an event by observing new information of the real world. It is a fundamental theorem in probability theory that allows updating probabilities based on new evidence. It provides a principled way to combine prior knowledge with new data to update the probabilities of different outcomes. Posterior probability is calculated by Bayes' Theorem. Bayes rule in AI is a mathematical formula that combines the prior probability, likelihood, and evidence to calculate the posterior probability.

Bayes' rule has been widely used in AI for classification, prediction, and decision-making tasks where uncertainty needs to be addressed.

Bayes' theorem can be derived using product rule and conditional probability of event A with known event B:

As from product rule, we can write:

$P(A \wedge B) = P(A|B)\, P(B)$

Similarly, the probability of event B with known event A:

$P(A \wedge B) = P(B|A)\, P(A)$

Equating right hand side of both the equations, we will get:

$$P(A|B) = \frac{P(B|A)\, P(A)}{P(B)} \quad \text{....(a)}$$

The above equation (a) is called as Bayes' rule or Bayes' theorem. This equation is basic of most modern AI systems for probabilistic inference.

It shows the simple relationship between joint and conditional probabilities.

Sometimes, you can treat P(B) as a normalization constant ‘α’.

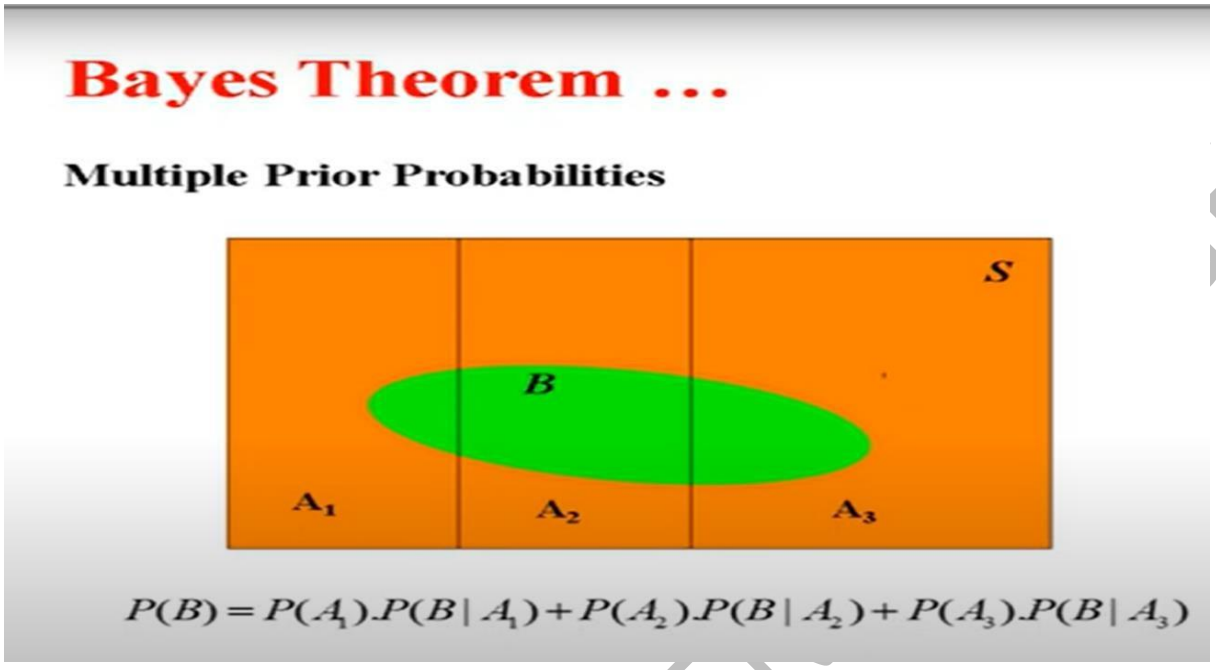
$P(A|\ B) = \alpha\ P(B\ |\ A)P(A)$

Here,

P(A|B) is known as posterior, which we need to calculate, and it will be read as Probability of hypothesis A when we have occurred an evidence B.

P(B|A) is called the likelihood, in which we consider that hypothesis is true, then we calculate the probability of evidence.

P(A) is called the prior probability, probability of hypothesis before considering the evidence P(B) is called marginal probability, pure probability of an evidence.



In the equation (a), in general, we can write P (B) = P(Ai)*P(B|Ai), hence the Bayes' rule can be written as:

$$P(A_i|B) = \frac{P(A_i)*P(B|A_i)}{\sum_{i=1}^{k} P(A_i)*P(B|A_i)}$$

Where A1, A2, A3,. , An is a set of mutually exclusive and exhaustive events.

Applying Bayes' rule:

Bayes' rule allows us to compute the single term P(B|A) in terms of P(A|B), P(B), and P(A). This is very useful in cases where we have a good probability of these three terms and want to determine the fourth one.

Suppose we want to perceive the effect of some unknown cause, and want to compute that cause, then the Bayes' rule becomes:

$$\mathbf{P(cause|effect)} = \frac{P(effect|cause)\,P(cause)}{P(effect)}$$

When is Bayes Rule Useful?

Sometimes it's easier to get P(X|Y) than P(Y|X).

Information is typically available in the form P(effect | cause) rather than P(cause | effect)

For example, P(symptom | disease) is easy to measure empirically but obtaining P(disease | symptom) is harder

Example:

As per statistics,

- Disease meningitis causes stiff neck 50% of the time.
- The proportion of persons having meningitis is 1/500.
- The proportion of people having stiff neck is 1/20.

Question: what is the percentage of people had disease meningitis and complain about stiff neck?

Solution:

We can calculate as;

- Let 'A' be the proposition that patient has stiff neck
- Let 'B' be the proposition that patient has meningitis
- we know that meningitis causes stiff neck 50% of the time, P(A| B)=0.5
- P(A)= 1/20
- P(B) = 1/500
- We know Bayes theorem:
- P(B|A) = P(A|B) * P(B) / P(A)

  = (0.5 * 1/500) / (1/20)

  = 0.02% (had meningitis and complain about stiff neck)

Application of Bayes' theorem in Artificial intelligence:

Following are some applications of Bayes' theorem:

- It is used to calculate the next step of the robot when the already executed step is given.
- Bayes' theorem is helpful in weather forecasting.
- It can solve the Monty Hall problem.

Bayesian Belief Network

Bayesian belief network is key computer technology for dealing with probabilistic events and to solve a problem which has uncertainty.

We can define a Bayesian network as:

- "A Bayesian network is a probabilistic graphical model which represents a set of variables (nodes) and their conditional dependencies using a Directed Acyclic Graph(DAG).“

It is also called a Bayes network, belief network, decision network, or Bayesian model.

Bayesian networks are probabilistic, because these networks are built from a probability distribution, and also use probability theory for prediction and anomaly detection.

Real world applications are probabilistic in nature, and to represent the relationship between multiple events, we need a Bayesian network.

It can also be used in various tasks including prediction, anomaly detection, diagnostics, automated insight, reasoning, time series prediction, and decision making under uncertainty.

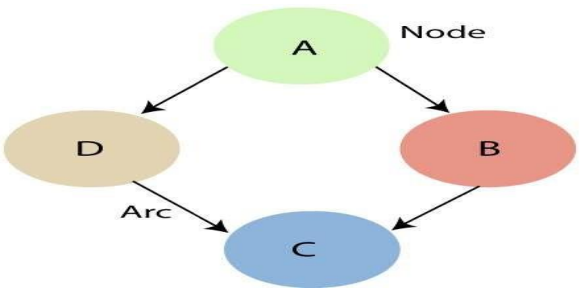
Bayesian Network can be used for building models from data and experts opinions

It consists of two parts:

- Directed Acyclic Graph
- Table of conditional probabilities.

The generalized form of Bayesian network that represents and solve decision problems under uncertain knowledge is known as an Influence diagram.

Directed Acyclic Graph



A Bayesian network graph is made up of nodes and Arcs (directed links), where:

Each node corresponds to the random variables, and a variable can be continuous or discrete.

Arc or directed arrows represent the causal relationship or conditional probabilities between random variables. These directed links or arrows connect the pair of nodes in the graph. These links represent that one node directly influence the other node, and if there is no directed link that means that nodes are independent with each other.

In the above diagram, A, B, C, and D are random variables represented by the nodes of the network graph. If we are considering node B, which is connected with node A by a directed arrow, then node A is called the parent of Node B. Node C is independent of node A.

Note: The Bayesian network graph does not contain any cyclic graph. Hence, it is known as a directed acyclic graph or DAG.

Conditional Probability Tables

- The conditional distributions for each node are given as conditional probabilities table or CPT.
- The conditional probability tables in the network give the probabilities for the value of the random variables depending on the combination of values for the parent nodes.
- Each row in the CPT must be sum to 1.
- All variables are Boolean, and therefore, the probability of a true value is p, the probability of false value must be 1 – p.
- In CPT, a boolean variable with k boolean parents contains 2K probabilities. Hence, if there are two parents, then CPT will contain 4 probability values
- A variable with no parents has only one row, representing prior probabilities of each possible values of the variable.

The Bayesian network has mainly two components:

- Causal Component
- Actual numbers

Each node in the Bayesian network has condition probability distribution P(Xi |Parent(Xi)), which determines the effect of the parent on that node.

Bayesian network is based on Joint probability distribution and conditional probability. So, let's first understand the joint probability distribution.

Joint probability distribution:

If we have variables x1, x2, x3,……. , xn, then the probabilities of a different combination of x1,

x2, x3.. xn, are known as Joint probability distribution.

P[x1, x2, x3,. , xn], it can be written as the following way in terms of the joint probability distribution.

P[x1, x2, x3,....., xn] = P[x1| x2, x3,....., xn]P[x2, x3,. , xn]

= P[x1| x2, x3,....., xn]P[x2|x3,....., xn] ….P[xn-1|xn]P[xn].

In general for each variable Xi, we can write the equation as: P(Xi|Xi-1,. , X1) = P(Xi |Parents(Xi))

Explanation of Bayesian network:

Let's understand the Bayesian network through an example by creating a directed acyclic graph:

Example: Harry installed a new burglar alarm at his home to detect burglary. The alarm reliably responds at detecting a burglary but also responds for minor earthquakes. Harry has two neighbors David and Sophia, who have taken a responsibility to inform Harry at work when they hear the alarm. David always calls Harry when he hears the alarm, but sometimes he got confused with the phone ringing and calls at that time too. On the other hand, Sophia likes to listen to high music, so sometimes she misses to hear the alarm. Here we would like to compute the probability of Burglary Alarm.

Problem:

Calculate the probability that alarm has sounded, but there is neither a burglary, nor an earthquake occurred, and David and Sophia both called the Harry.

Solution:

- The Bayesian network for the above problem is given below. The network structure is showing that burglary and earthquake is the parent node of the alarm and directly affecting the probability of alarm's going off, but David and Sophia's calls depend on alarm probability.
- The network is representing that our assumptions do not directly perceive the burglary and also do not notice the minor earthquake, and they also not confer before calling.

List of all events occurring in this network:

- Burglary (B)
- Earthquake(E)
- Alarm(A)
- David Calls(D)
- Sophia calls(S)

We can write the events of problem statement in the form of probability:

P[D, S, A, B, E], can rewrite the above probability statement using joint probability distribution:

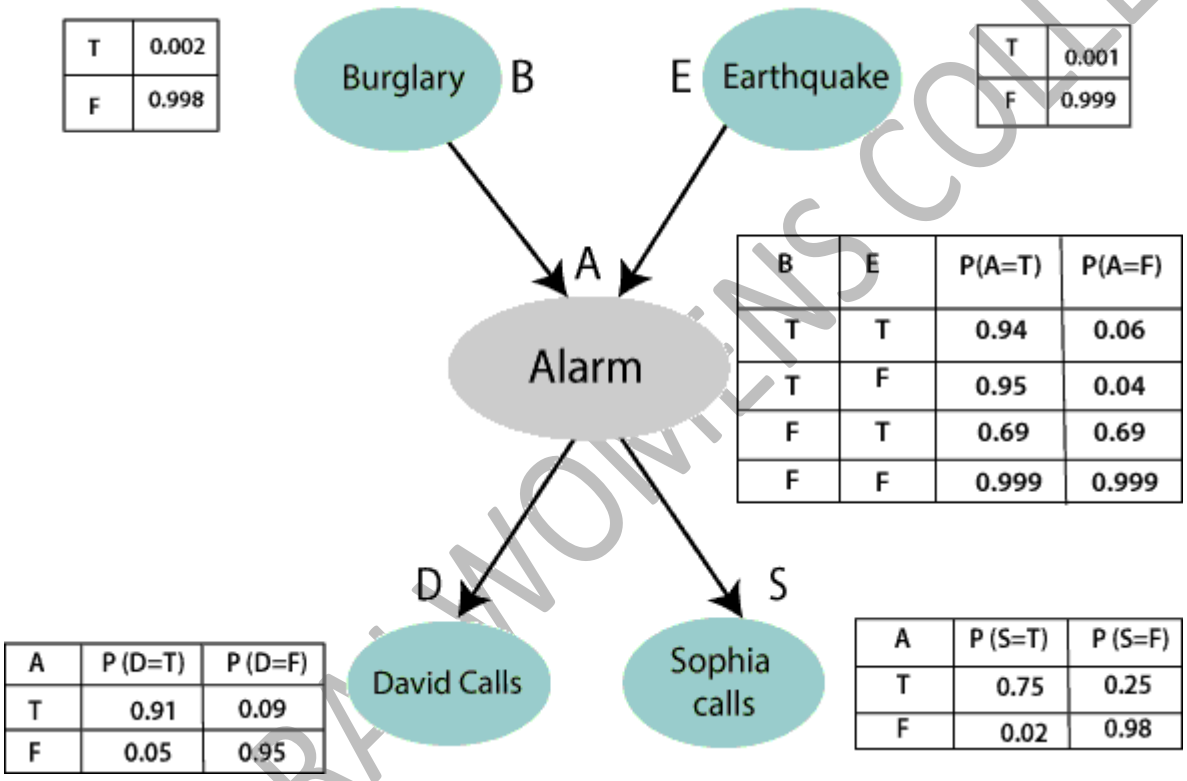
P[D, S, A, B, E] = P[D | S, A, B, E]. P[S, A, B, E]

= P[D | S, A, B, E]. P[S | A, B, E]. P[A, B, E]

= P [D| A]. P [S| A, B, E]. P[A, B, E]

= P[D | A]. P[S | A]. P[A| B, E]. P[B, E]

= P[D | A]. P[S | A]. P[A| B, E]. P[B |E]. P[E]



T	0.002
F	0.998

T	0.001
F	0.999

B	E	P(A=T)	P(A=F)
T	T	0.94	0.06
T	F	0.95	0.04
F	T	0.69	0.69
F	F	0.999	0.999

A	P (D=T)	P (D=F)
T	0.91	0.09
F	0.05	0.95

A	P (S=T)	P (S=F)
T	0.75	0.25
F	0.02	0.98

Let's take the observed probability for the Burglary and earthquake component:

P(B= True) = 0.002, which is the probability of burglary.

P(B= False)= 0.998, which is the probability of no burglary.

P(E= True)= 0.001, which is the probability of a minor earthquake

P(E= False)= 0.999, Which is the probability that an earthquake not occurred.

We can provide the conditional probabilities as per the below tables:

Conditional probability table for Alarm A:

The Conditional probability of Alarm A depends on Burglar and earthquake:

B	E	P(A= True)	P(A= False)
True	True	0.94	0.06
True	False	0.95	0.04
False	True	0.31	0.69
False	False	0.001	0.999

Conditional probability table for David Calls:

The Conditional probability of David that he will call depends on the probability of Alarm.

A	P(D= True)	P(D= False)
True	0.91	0.09
False	0.05	0.95

Conditional probability table for Sophia Calls:

The Conditional probability of Sophia that she calls is depending on its Parent Node "Alarm."

A	P(S= True)	P(S= False)
True	0.75	0.25
False	0.02	0.98

From the formula of joint distribution, we can write the problem statement in the form of probability distribution:

P(S, D, A, ¬B, ¬E) = P (S|A) *P (D|A)*P (A|¬B ^ ¬E) *P (¬B) *P (¬E).

= 0.75* 0.91* 0.001* 0.998*0.999

= 0.00068045.

Hence, a Bayesian network can answer any query about the domain by using Joint distribution.

The semantics of Bayesian Network:

There are two ways to understand the semantics of the Bayesian network, which is given below:

1. To understand the network as the representation of the Joint probability distribution.

   It is helpful to understand how to construct the network.

2. To understand the network as an encoding of a collection of conditional independence statements.

   It is helpful in designing inference procedure.

Rules of Inference in Artificial intelligence Inference:

In artificial intelligence, we need intelligent computers which can create new logic from old logic or by evidence, so generating the conclusions from evidence and facts is termed as Inference.

Inference rules:

Inference rules are the templates for generating valid arguments. Inference rules are applied to derive proofs in artificial intelligence, and the proof is a sequence of the conclusion that leads to the desired goal.

In inference rules, the implication among all the connectives plays an important role.

Following are some terminologies related to inference rules:

- **Implication:** It is one of the logical connectives which can be represented as P → Q. It is a Boolean expression.
- **Converse:** The converse of implication, which means the right-hand side proposition goes to the left-hand side and vice-versa. It can be written as Q → P.
- **Contrapositive:** The negation of converse is termed as contrapositive, and it can be represented as ¬ Q → ¬ P.
- **Inverse:** The negation of implication is called inverse. It can be represented as ¬ P → ¬ Q.

From the above term some of the compound statements are equivalent to each other, which we can prove using truth table:

Types of Inference rules:

1. Modus Ponens:

The Modus Ponens rule is one of the most important rules of inference, and it states that if P and P → Q is true, then we can infer that Q will be true. It can be represented as:

Notation for Modus ponens: $\frac{P \rightarrow Q, \quad P}{\therefore Q}$

Example:

Statement-1: "If I am sleepy then I go to bed" ==> P→ Q

Statement-2: "I am sleepy" ==> P Conclusion: "I go to bed." ==> Q.

Hence, we can say that, if P→ Q is true and P is true then Q will be true.

Proof by Truth table:

P	Q	$P \rightarrow Q$
0	**0**	**0**
0	**1**	**1**
1	**0**	**0**
1	**1**	**1** ←

2. Modus Tollens:

The Modus Tollens rule state that if P→ Q is true and ¬ Q is true, then ¬ P will also true. It can be represented as:

Notation for Modus Tollens: $\frac{P \rightarrow Q, \quad \sim Q}{\sim P}$

Statement-1: "If I am sleepy then I go to bed" ==> P→Q

Statement-2: "I do not go to the bed."==> ~Q

Statement-3: Which infers that "I am not sleepy" => ~P

Proof by Truth table:

P	Q	$\sim P$	$\sim Q$	$P \rightarrow Q$
0	**0**	**1**	**1**	**1** ←
0	**1**	**1**	**0**	**1**
1	**0**	**0**	**1**	**0**
1	**1**	**0**	**0**	**1**

3. Hypothetical Syllogism:

The Hypothetical Syllogism rule state that if P→R is true whenever P→Q is true, and Q→R is true. It can be represented as the following notation:

Example:

Statement-1: If you have my home key then you can unlock my home. P→Q

Statement-2: If you can unlock my home then you can take my money. Q→R

Conclusion: If you have my home key then you can take my money. P→R

Proof by truth table:

P	Q	R	$P \rightarrow Q$	$Q \rightarrow R$	$P \rightarrow R$
0	0	0	1	1	1
0	0	1	1	1	1
0	1	0	1	0	1
0	1	1	1	1	1
1	0	0	0	1	1
1	0	1	0	1	1
1	1	0	1	0	0
1	1	1	1	1	1

4. Disjunctive Syllogism:

The Disjunctive syllogism rule state that if PvQ is true, and ¬P is true, then Q will be true. It can be represented as:

Notation of Disjunctive syllogism: $\frac{P \vee Q,\ \neg P}{Q}$

Example:

Statement-1: Today is Sunday or Monday. ==>PvQ

Statement-2: Today is not Sunday. ==> ¬P Conclusion: Today is Monday. ==> Q

Proof by truth-table:

P	Q	$\neg P$	$P \vee Q$
0	0	1	0
0	1	1	1
1	0	0	1
1	1	0	1

5. Addition:

The Addition rule is one the common inference rule, and it states that If P is true, then PvQ will be true.

Notation of Addition: $\frac{P}{P \vee Q}$

Example:

Statement: I have a vanilla ice-cream. ==> P

Statement-2: I have Chocolate ice-cream.

Conclusion: I have vanilla or chocolate ice-cream. ==> (PvQ)

Proof by Truth-Table:

P	Q	$P \vee Q$	
0	0	0	
1	0	1	←
0	1	1	
1	1	1	←

Simplification:

The simplification rule state that if PA Q is true, then Q or P will also be true. It can be represented as:

Notation of Simplification rule: $\frac{P \wedge Q}{Q}$ Or $\frac{P \wedge Q}{P}$

Proof by Truth-Table:

P	Q	$P \wedge Q$	
0	0	0	
1	0	0	
0	1	0	
1	1	1	←

7. Resolution:

The Resolution rule state that if PvQ and ¬ PAR is true, then QvR will also be true. It can be represented as

Notation of Resolution $\frac{P \vee Q,\ \neg P \wedge R}{Q \vee R}$

Proof by Truth-Table:

P	¬ P	Q	R	$P \vee Q$	¬ P∧R	$Q \vee R$	
0	1	0	0	0	0	0	
0	1	0	1	0	0	1	
0	1	1	0	1	1	1	←
0	1	1	1	1	1	1	←
1	0	0	0	1	0	0	
1	0	0	1	1	0	1	
1	0	1	0	1	0	1	
1	0	1	1	1	0	1	←

Inference in Bayesian Network

Purpose:

- Probabilistic Inference System is to compute Posterior Probability Distribution for a set of query variables, given some observed event.
- That is, some assignment of values to a set of evidence variables.

Notations:

- X – Denotes the query variable
- E – Set of Evidence variables
- e – Particular observed event
- Y – Non evidence, non query variable Y1, …., Yn(Hidden variables)
- The complete set of variables X = {X} U E U Y
- A typical query ask for the Posterior Probability Distribution P{X|e}

Types of Inferences:

- Inference by Enumeration
  - (inference by listing or recording all variables)
- Inference by Variable Elimination
  - (inference by variable removal)

Example: In burglary network, calculate P(B | J,M)

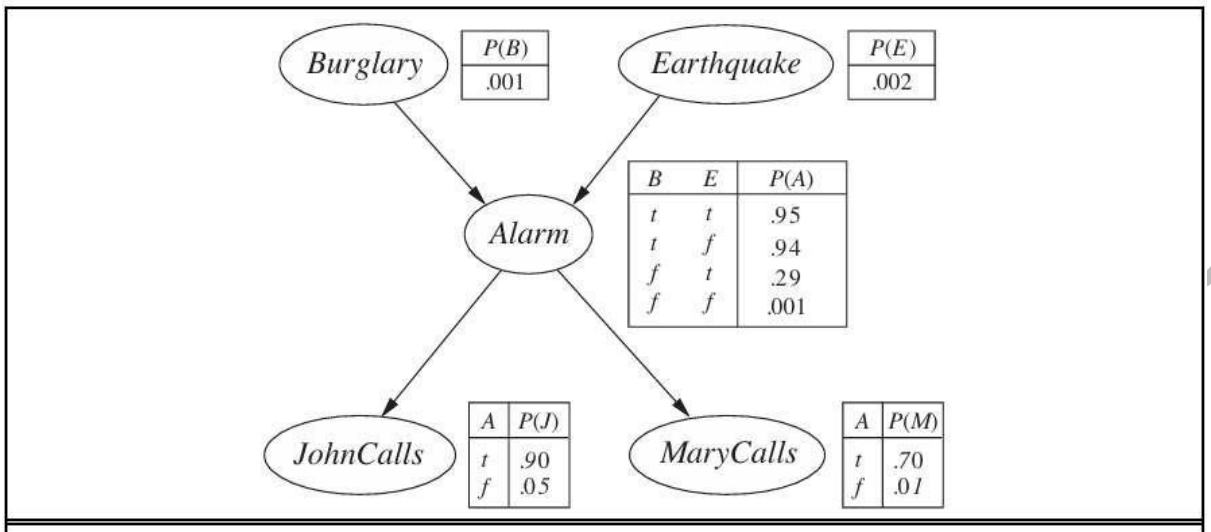


Figure 14.2 A typical Bayesian network, showing both the topology and the conditional probability tables (CPTs). In the CPTs, the letters B, E, A, J, and M stand for *Burglary*, *Earthquake*, *Alarm*, *JohnCalls*, and *MaryCalls*, respectively.

In the burglary network, we might observe the event in which

- JohnCalls = true and MaryCalls=true

We could then ask for, say, the probability that a burglary has occurred:

- P(Burglary|JohnCalls = true, MaryCalls = true) = ?
- Burglary – query variable (X)
- JohnCalls – Evidence Variable 1 (E1)
- MaryCalls – Evidence Variable 2 (E2)
- The hidden variables of this query are earthquake and alarm

Inference by Enumeration

Any conditional probability can be computed by summing terms from the full joint distribution. More specifically, a query P(X|e) can be answered using equation:

- P(X|e) = α P(X,e) = α Σy P(X,e,y)
    - Where α is normalized constant
    - X – Query variable
    - e – event
    - y – number of terms(hidden variables)

Example: P(B|j,m) = P(B,j,m)

P(j,m)

- So, treating P(j,m) as constant values, we can write as;
- P(b | j,m) = α P(b,j,m)

= α Σ E,A P(b,e,a,j,m)

= α Σ E,A P(b) P(e) P(a|e,b) P(j|a) P(m|a)

= α P(b) Σ E P(e) Σ A P(A|e,b) P(j|a) P(m|a)

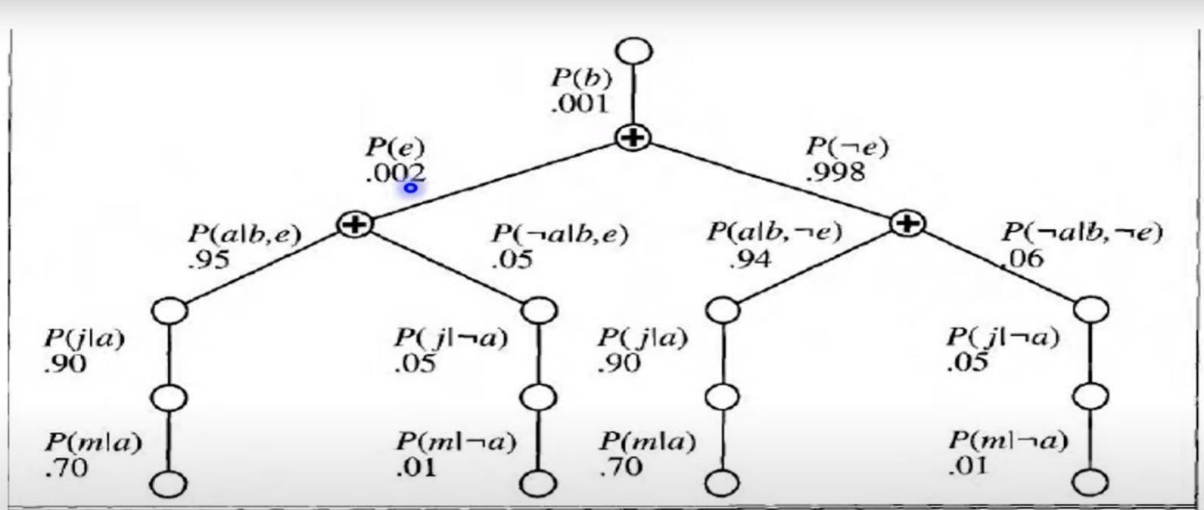


Figure 14.8 The structure of the expression shown in Equation (14.3). The evaluation proceeds top-down, multiplying values along each path and summing at the "+" nodes. Notice the repetition of the paths for j and m.

This expression can be evaluated by looping through the variables in order, multiplying CPT entries as we go. For each summation, we also need to loop over the variable's possible values. The structure of this computation is shown in Figure 14.8. Using the numbers, we obtain P(b|j,m)=α×0.00059224.

The corresponding computation for ¬b yields α×0.0014919; hence, P(B|j,m)=α(0.00059224,0.0014919) ≈ 0.284,0.716 .

That is, the chance of a burglary, given calls from both neighbors, is about 28%

Drawback

- Note that the tree in Figure 14.8 makes explicit the repeated subexpressions evaluated by the algorithm.
- The products P(j |a)P(m|a) and P(j |¬a)P(m|¬a) are computed twice, once for each value of e.
- The next section describes a general method that avoids such wasted computations

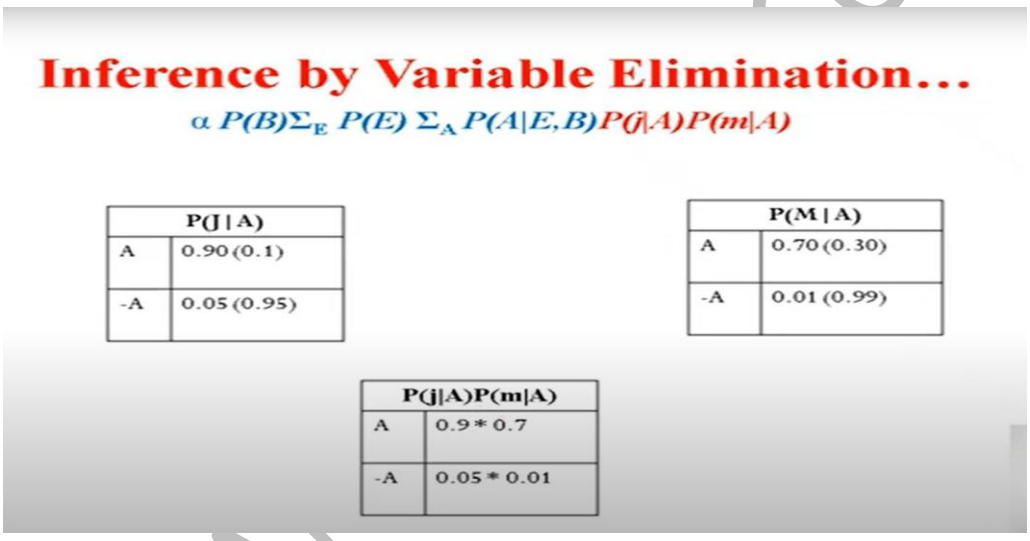
Inference by Variable Elimination

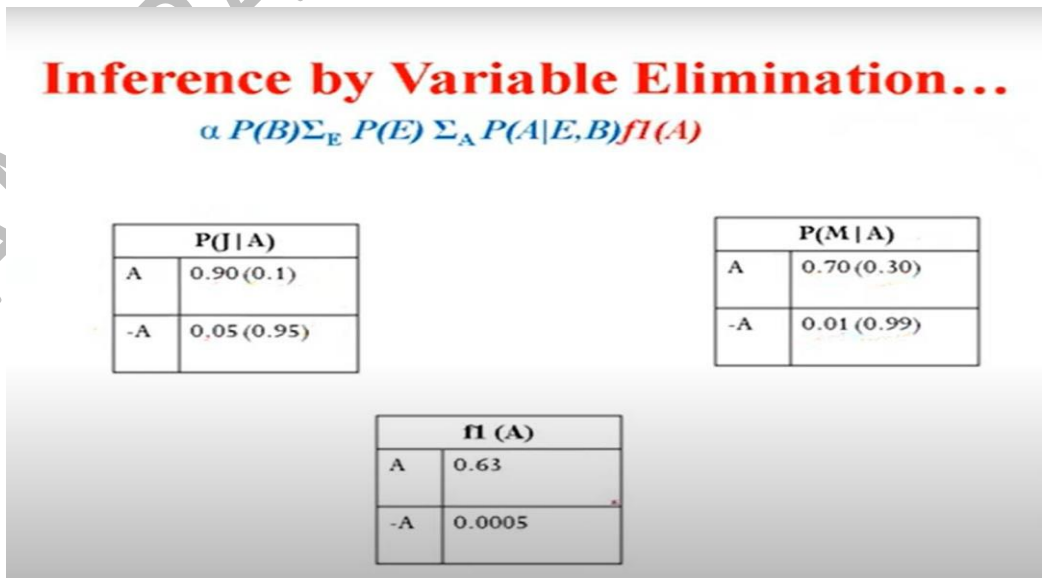
The enumeration algorithm can be improved substantially by eliminating repeated calculations. The idea is simple: do the calculation once and save the results for later use. This is a form of dynamic programming. Variable elimination works by evaluating expressions, from the previous equation (derived in inference by enumeration)

Intermediate results are stored, and summations over each variable are done only for those portions of the expression that depend on the variable.

Let us illustrate this process for the burglary network. We evaluate the below expression in such a way that we annotated each part of the expression with the name of the associated variable; these parts are called factors.

- P(b|j,m) = α P(b) Σe P(e) Σa P(a|e,b) P(j|a) P(m|a)





Inference by Variable Elimination...

$\alpha\ P(B)\Sigma_E\ P(E)\ \Sigma_A P(A|E,B)f1(A)$

f1 (A)	
A	0.63
-A	0.0005

P(A | E,B)	
e , b	0.95 (0.05)
e , -b	0.29 (0.71)
-e , b	0.94(0.06)
-e , -b	0.001(0.999)

Σ_AP(A | E,B) f1(A)	
e , b	0.95 * 0.63 + 0.05 * 0.0005
e , -b	0.29 *0.63 + 0.71 * 0.0005
-e , b	0.94 * 0.63 + 0.06 * 0.0005
-e , -b	0.001 * 0.63 + 0.999 * 0.0005

Inference by Variable Elimination...

$\alpha\ P(B)\Sigma_E\ P(E)\ f2(E,B)$

f1 (A)	
A	0.63
-A	0.0005

P(A|E,B)	
e , b	0.95 (0.05)
e , -b	0.29 (0.71)
-e , b	0.94(0.06)
-e , -b	0.001(0.999)

f2(E,B)	
e , b	0.60
e , -b	0.18
-e , b	0.59
-e , -b	0.001

Inference by Variable Elimination...

$\alpha\ P(B)\Sigma_E\ P(E)\ f2(E,B)$

P(E=T)	P(E=F)
0.002	0.998

P(B=T)	P(B=F)
0.001	0.999

f2(E,B)	
e , b	0.60
e , -b	0.18
-e , b	0.59
-e , -b	0.001

$P(B)\Sigma_E\ P(E)\ f2(E,B)$	
b	0.60 * 0.002 * 0.001 + 0.59 * 0.998 * 0.001
-b	0.18 * 0.002 * 0.999 + 0.001 * 0.998 * 0.999

Inference by Variable Elimination...

$\alpha\ f3(B)$

P(E=T)	P(E=F)
0.002	0.998

P(B=T)	P(B=F)
0.001	0.999

f2(E,B)	
e , b	0.60
e , -b	0.18
-e , b	0.59
-e , -b	0.001

f3(B)	
b	0.0006
-b	0.0013

Inference by Variable Elimination...

$\alpha f3(B) \rightarrow P(B \mid j, m)$

f3(B)	
b	0.0006
-b	0.0013

$N = 0.0006 + 0.0013 = 0.0019$

P(B | j, m)	
b	0.32
-b	0.68

That is, the chance of a burglary, given calls from both neighbors, is about 32%.

Temporal Models (Probabilistic reasoning over time)

Topics

- Time and Uncertainty
    - States and observations
    - Stationary processes and the Markov assumption
- Inference in Temporal Model

Introduction

Agents in uncertain environments must be able to keep track of the current state of the environment, just as logical agents must. But this is difficult by partial and noisy data, because the environment is uncertain over time. At best, the agent will be able to obtain only a probabilistic assessment of the current situation.

Till now we have learned techniques for probabilistic reasoning in the context of static worlds, in which each random variable has a single fixed value. For example, when repairing a car, we assume that whatever is broken remains broken during the process of diagnosis; our job is to infer the state of the car from observed evidence, which also remains fixed.

Temporal Models - Time and Uncertainty

A changing world is modelled using a random variable for each aspect of the environment state, at each point in time. The relations among these variables describe how the state evolves.

Now consider a slightly different problem: treating a diabetic patient. Here, we have evidence such as, recent insulin doses, food intake, blood sugar measurements, and other physical signs. The task is to assess the current state of the patient, including the actual blood sugar level and insulin level.

Why Temporal model?

Given this information, the doctor (or patient) makes a decision about the patient's food intake and insulin dose. Here, The dynamic aspects of the problem are essential. Blood sugar levels and measurements thereof can change rapidly over time, depending on one's recent food intake and insulin doses, one's metabolic activity, the time of day, and so on.

To assess the current state from the history of evidence and to predict the outcomes of treatment actions, we must model these changes. In order to handle such dynamic aspects in probabilistic reasoning we require a special kind of model called as temporal model which assess the probabilistic reasoning over time.

It is a way to reference data in time and query the state of an entity at a given time. They can also specify the temporal-dependent relation among collaborative activities.

Temporal models are models created to evaluate and forecast time-series data. Data that is gathered over time, such as stock prices, weather patterns, or website traffic patterns, is referred to as time-series data. In order to generate predictions or forecast future events, temporal models try to identify patterns and trends in this data throughout time. Due to their capacity to predict future events using historical data, temporal models have gained in prominence recently.

They have a wide range of uses, including estimating future sales for a company, predicting weather patterns, and spotting irregularities in network traffic.

States and Observations

The process of change can be viewed as a series of snapshots, or time slices, describes the state of the world at a particular time, t. Each snapshot, contains a set of random variables, some of which are observable and some not. Xt denote set of unobservable state variable at time t, and Et denote set of observable evidence variables. The observation at time t is Et =et for some set of values et.

Example: Umbrella and Rain. Suppose you are a security guard for some secret underground installation. You want to know whether it is raining today. But, your only access to the outside world occurs each morning, when you see the director coming in with, or without an umbrella.

For each day t, the set Et (observable evidence variables) thus contains a single evidence variable Umbrellat or Ut for short (whether the umbrella appears) and the set Xt (unobservable state variable) contains a single state variable Raint or Rt for short (whether it is raining or not) Hence we can assume Et =Ut and Xt = Rt and if Et = true then Xt = true i.e., if Ut = true then Rt = true.

Other problems involve larger sets of variables, in diabetes example, we might have evidence variables, such as MeasuredBloodSugart and PulseRatet, and state variables, such as BloodSugart and StomachContentst. (Notice that BloodSugart and MeasuredBloodSugart are not the same variable; this is how we deal with noisy measurements of actual quantities.)

The interval between time slices also depends on the problem. For diabetes monitoring, a suitable interval might be an hour rather than a day.

We generally assume a fixed, finite interval; this means that times can be labelled by integers. We will assume that the state sequence starts at t = 0 and evidence starts arriving at t = 1 rather than t = 0. Hence, our umbrella world is represented by state variables R0, R1, R2, … and evidence variables U1,U2,…. We will use the notation a:b to denote the sequence of integers from a to b and the notation Xa:b to denote the corresponding set of variables from Xa to Xb. For example, U1:3 corresponds to the variables U1,U2,U3.

In states and observations, we are going to predict the environment only by using the current evidence variable, but not from the historical values of those evidence variable i.e., if umbrella present then raining outside. Current situation will be predicted only by the current evidence

Transition and sensor models

With the set of state and evidence variables for a given problem, the next step is to specify how the world evolves (the transition model) and how the evidence variables get their values (the sensor model). The transition model specifies the probability distribution over the latest state variables, given the previous values, that is, P(Xt |X0:t−1).

We face a problem: the set X0:t−1 is unbounded in size, as t increases. We solve the problem: by making a Markov assumption.

Markov assumption

Markov assumption– the current state depends on only a finite fixed number of previous states. Processes satisfying this assumption were first studied in depth by the Russian statistician Andrei Markov (1856–1922) and are called Markov processes or Markov chains.

There are various types:

First-order Markov process

The simplest is the first-order Markov process, in which the current state depends only on the previous state and not on any earlier states. Using our notation, the corresponding conditional independence assertion states that, for all t,

P(Xt|X0:t-1) = P(Xt| Xt-1)

Hence, in a first-order Markov process, the transition model is the conditional distribution P(Xt

|Xt−1) (Markov Property). Any chain following Markov Property is Markov Chain.

Second-order Markov process

In second-order Markov process, the current state depends on previous two states. The transition model for a second-order Markov process is the conditional distribution P (Xt| Xt-2, Xt-1).

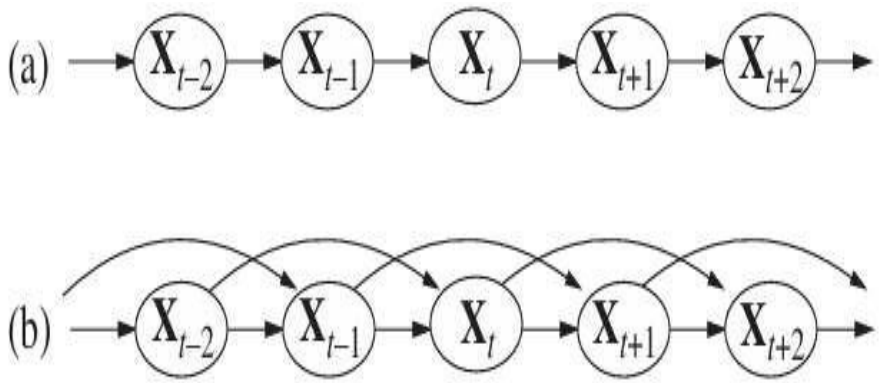


Figure 15.1 (a) Bayesian network structure corresponding to a first-order Markov process with state defined by the variables $\mathbf{X}_t$. (b) A second-order Markov process.

Stationary process

Even with the Markov assumption there is still a problem: there are infinitely many possible values of t. We avoid this problem by assuming that changes in the world state are caused by a stationary process—that is, a process of change that is governed by laws that do not themselves change over time.

In the umbrella world, then, the conditional probability that the umbrella appears, P(Ut | Parents(Ut)), is the same for all t. The structure in Figure is a first-order Markov process—the probability of rain is assumed to depend only on whether it rained the previous day.

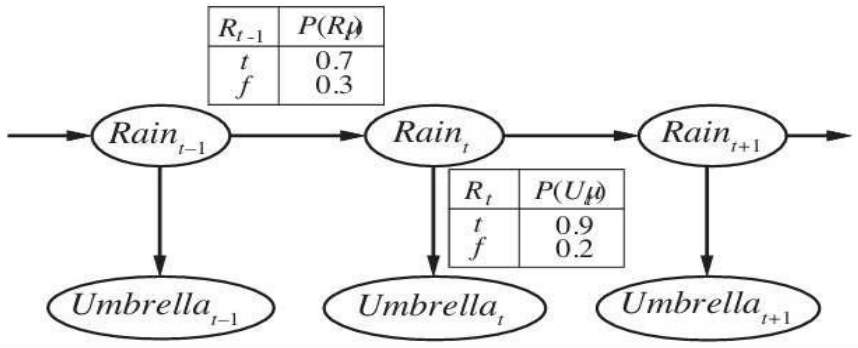


Figure 15.2 Bayesian network structure and conditional distributions describing the umbrella world. The transition model is $P(Rain_t \mid Rain_{t-1})$ and the sensor model is $P(Umbrella_t \mid Rain_t)$.

Inference in Temporal Model

Basic inference tasks:

- Filtering or monitoring
- Prediction
- Smoothing
- Most likely explanation

Filtering

Maintain the current state estimate and update it. Need to go back over the entire history of percepts for each updates. In the umbrella example, computing the probability of rain today, given all the observations of the umbrella.

- P(Xt+1 | e1:t+1) = f(et+1, P(Xt | e1:t))
- Where Xt = Rt and et = Ut

In our example, we wish to compute P(Xt |e1:t). This is the task of computing the belief state—the posterior distribution over the most recent state—given all evidence to date. Filtering is also called state estimation. Filtering is, a rational agent needs to do in order to keep track of the current state so that rational decisions can be made.

Prediction

This is the task of computing the posterior distribution over the future state given all evidence. In the umbrella example, this might mean computing the probability of rain three days from now, given all the observations to date. Prediction is useful for evaluating possible courses of action.

- P(Xt+k | e1:t) for some k>0. (Here k=3).

Smoothing

This is the task of computing the distribution over past states, given evidence up to the present state. In the umbrella example, computing the probability that, it rained last Wednesday, given all the observations of the umbrella carrier made up to today. That is, we wish to compute,

- P(Xk | e1:t) for some k such that 0 ≤ k <t

Most likely explanation

Given a sequence of observations, To compute the sequence of states that is most likely to have generated those observations. If the umbrella appears on each of the first three days and is absent on the fourth, then the most likely explanation is that it rained on the first three days and did not rain on the fourth, and again umbrella appears in fifth day. Does the absence of umbrella in fourth day means that, it was not raining or the director forgot to bring it? (continuously 3 days

raining and 5th day is also raining, hence fourth day not raining is not acceptable). The director forgot to bring the umbrella is most likely explanation.

Markov Model

The Markov Model is stochastic model for randomly changing systems. It's assumed that the future states don't depend on the past states, rather they are determined only by the current state. This model shows all possible states and transition rate and probabilities between them.

Hidden Model

In Hidden Models, we don't have idea on which state matches which physical event instead each state matches the given output. We observe output over time to determine the sequence of possible states.

Hidden Markov Model

Hidden + Markov Model

A statistical model called a Hidden Markov Model (HMM) is used to describe systems with changing unobservable states over time. It is used to describe the probabilistic relationship between a sequence of observations and a sequence of hidden states.

It is used to predict future observations or classify sequences, based on the underlying hidden process that generates the data.

An HMM consists of two types of variables: hidden states and observations. The hidden states are the underlying variables that generate the observed data, but they are not directly observable. The observations are the variables that are measured and observed.

The relationship between the hidden states and the observations is modelled using a probability distribution. The Hidden Markov Model (HMM) is the relationship between the hidden states and the observations using two sets of probabilities: the transition probabilities and the emission probabilities. The transition probabilities describe the probability of transitioning from one hidden state to another. The emission probabilities describe the probability of observing an output given a hidden state.

HMM is a statistical model, in which the system being modelled, is assumed to be a Markov process (Memory less process: its future and past are independent) with hidden states.

- Has a set of states each of which has limited number of transitions and emissions.
- Each transition between states has an assigned probability.
- Each model starts from start state and ends in end state.

The Hidden Markov model is a probabilistic model which is used to explain or derive the probabilistic characteristic of any random process. It basically says that an observed event will not be corresponding to its step-by-step status but related to a set of probability distributions.

Let's assume a system that is being modelled is assumed to be a Markov chain and in the process, there are some hidden states. In that case, we can say that hidden states are a process that depends on the main Markov process/chain.

The main goal of HMM is to learn about a Markov chain by observing its hidden states. Considering a Markov process X with hidden states Y here the HMM solidifies that for each time stamp the probability distribution of Y must not depend on the history of X according to that time.

Hidden Markov Model with an Example

To explain it more we can take the example of two friends, Rahul and Ashok. Now Rahul completes his daily life works according to the weather conditions. Major three activities completed by Rahul are- go jogging, go to the office, and cleaning his residence. What Rahul is doing today depends on whether and whatever Rahul does he tells Ashok and Ashok has no proper information about the weather But Ashok can assume the weather condition according to Rahul work.

Ashok believes that the weather operates as a discrete Markov chain, wherein the chain there are only two states whether the weather is Rainy or it is sunny. The condition of the weather cannot be observed by Ashok, here the conditions of the weather are hidden from Ashok. On

each day, there is a certain chance that Bob will perform one activity from the set of the following activities {"jog", "work"," clean"}, which are depending on the weather. Since Rahul tells Ashok that what he has done, those are the observations. The entire system is that of a hidden Markov model (HMM).

Here we can say that the parameter of HMM is known to Ashok because he has general information about the weather and he also knows what Rahul likes to do on average.

So let's consider a day where Rahul called Ashok and told him that he has cleaned his residence. In that scenario, Ashok will have a belief that there are more chances of a rainy day and we can say that belief Ashok has is the start probability of HMM let's say which is like the following.

The states and observation are:

states = ('Rainy', 'Sunny')

observations = ('walk', 'shop', 'clean')

And the start probability is:

start probability = {'Rainy': 0.6, 'Sunny': 0.4}

Now the distribution of the probability has the weightage more on the rainy day stateside so we can say there will be more chances for a day to being rainy again and the probabilities for next day weather states are as following

transition probability = {

'Rainy' : {'Rainy': 0.7, 'Sunny': 0.3},

'Sunny' : {'Rainy': 0.4, 'Sunny': 0.6},

}

From the above we can say the changes in the probability for a day is transition probabilities and according to the transition probability the emitted results for the probability of work that Rahul will perform is

emission probability = {

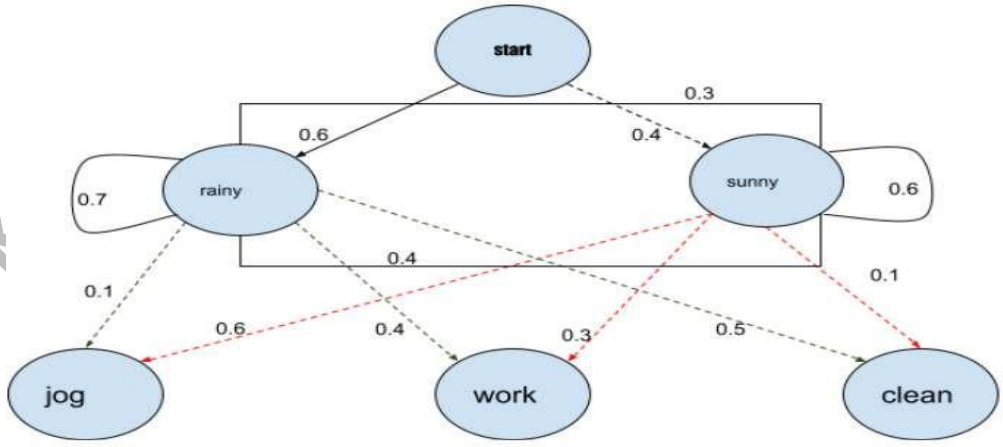
'Rainy' : {'jog': 0.1, 'work': 0.4, 'clean': 0.5},

'Sunny' : {'jog': 0.6, 'work: 0.3, 'clean': 0.1},

}

This probability can be considered as the emission probability. Using the emission probability Ashok can predict the states of the weather or using the transition probabilities Ashok can predict the work which Rahul is going to perform the next day.

Below image shown the HMM process for making probabilities



So here from the above intuition and the example we can understand how we can use this probabilistic model to make a prediction. Now let's just discuss the applications where it can be used.

Application of Hidden Markov Model

An application, where HMM is used, aims to recover the data sequence where the next sequence of the data cannot be observed immediately but the next data depends on the old sequences. Taking the above intuition into account the HMM can be used in the following applications:

- Computational finance speed analysis
- Speech recognition Speech synthesis
- Part-of-speech tagging
- Document separation in scanning solutions Machine translation
- Handwriting recognition
- Time series analysis
- Activity recognition
- Sequence classification
- Transportation forecasting

Hidden Markov Models in NLP

From the above application of HMM, we can understand that the applications where the HMM can be used have sequential data like time series data, audio, and video data, and text data or NLP data. In this article, our main focus is on those applications of NLP where we can use the HMM for better performance of the model, and here in the above-given list, we can see that one of the applications of the HMM is that we can use it in the Part-of-Speech tagging.

Three Basic Problems:

Three basic problems can be solved with Hidden Markov Models:

1. Evaluation Problem.
2. Decoding Problem.
3. Learning Problem or Optimization Problem.

Evaluation problem: Given the model, compute the probability that a particular output sequence was produced by that model (solved by the forward algorithm).

Decoding problem: Given the model, find the most likely sequence of hidden states which could have generated a given output sequence (solved by the Viterbi algorithm).

Learning problem: Given a set of output sequences, find the most likely set of state transition and output probabilities (solved by the Baum-Welch algorithm).

UNIT - IV

Reinforcement Learning (RL)

Reinforcement Learning is a branch of Machine Learning in which an agent learns by interacting with an environment. The agent performs actions, observes results, receives rewards, and improves its behaviour to maximize long-term reward.

Unlike supervised learning, RL does not require labelled data. Learning happens through trial and error.

Basic Terminology

Agent

The learner or decision-making entity.

Environment

The external system with which the agent interacts.

State (S)

A representation of the current situation of the agent.

Action (A)

All possible moves the agent can make.

Reward (R)

Numerical feedback received after performing an action.

Policy (π)

A mapping from states to actions.

Utility / Value

Measure of long-term desirability of a state.

Discount Factor (γ)

Controls the importance of future rewards.

Passive Reinforcement Learning in AI

A basic idea in artificial intelligence, Passive RL is the learning process of reaching a specific goal without doing exploratory actions unlike other RL techniques. The use of this form of RL may be applicable where the exploration is too expensive or poses certain risks and this is true in some branches of healthcare, finance and operations.

What does Passive Reinforcement Learning mean?

In the conventional reinforcement learning, then agents' purposefully search for possible state-action pairs in order to achieve the anticipated greatest cumulative rewards. However, in passive reinforcement learning, the agent's task is simplified: it receives a fixed policy (a pre-determined list of actions) but it learns how to assign a value to circles stating whether or not to follow the policy. The focus shifts from learning the optimal actions to estimating the reward for adhering to a given strategy. This is often used in cases where policies are generated externally and exploring alternative actions might be too complex, dangerous, or unnecessary.

Key Features of Passive Reinforcement Learning

Here are key features of Passive Reinforcement Learning:

1. **Policy Evaluation Focus:** It evaluates a fixed policy rather than exploring new actions.
2. **State Value Estimation:** Uses state values to estimate rewards over time following a predefined policy.
3. **Bellman Equation:** Applies the Bellman Equation iteratively to refine reward estimates.
4. **Model-Based and Model-Free:** Supports both approaches; model-based uses environment dynamics, while model-free learns from real experiences.
5. **Temporal Difference Learning:** Updates state values in real-time based on reward discrepancies.
6. **Low-Risk Application:** Suitable for scenarios where exploration is costly or risky, such as finance and healthcare.

Key Concepts in Passive Reinforcement Learning

1. **Policy Evaluation:** Since the policy is fixed, the primary objective is to evaluate this policy's efficacy. Policy evaluation determines how much reward an agent can expect over time by following a particular policy from any given state. The agent learns to predict future rewards by calculating the state values-an estimation of the total reward from each state while following the policy.
2. **Bellman Equation for Evaluation:** Passive RL employs the Bellman Equation to estimate state values. This recursive formula calculates the value of a state based on the immediate reward and the value of future states. The agent iteratively applies this equation to refine its understanding of the rewards associated with different states until it reaches an optimal estimate for each state.

3. **Model-Based and Model-Free Approaches**

   a. **Model-Based:** This approach assumes access to a model of the environment that includes the probabilities of transitioning between states and receiving rewards. In passive RL, this allows for dynamic programming techniques like policy iteration or value iteration.

   b. **Model-Free:** Model-free passive RL doesn't require knowledge of the environment's dynamics. Instead, it learns from actual experiences using techniques like Monte Carlo methods, which average out rewards from episodes, or Temporal Difference (TD) learning, which updates estimates based on discrepancies between consecutive steps.

Learning Methods in Passive RL

1. **Temporal Difference (TD) Learning:** TD learning is a popular approach in passive RL, especially in model-free scenarios. It updates the estimated state value after each interaction based on the difference between predicted and actual rewards (the temporal difference error). TD learning is particularly efficient because it can update values in real-time without waiting for an entire episode to conclude.
2. **Monte Carlo Methods:** Monte Carlo methods are episodic, meaning they wait until an episode ends to update state values based on accumulated rewards. This approach is less suitable for applications needing immediate feedback but is effective in deterministic or fully observable environments.

Some Applications of Passive RL

Passive reinforcement learning is widely applied in scenarios where agents must learn from observation or simulated experiences. For instance:

1. **Healthcare:** In fields like predictive diagnostics or medical decision support, passive RL can analyse historical data, learn the outcomes of fixed treatment strategies, and help improve future predictions.
2. **Finance:** Passive RL can model trading policies or risk management strategies by learning from historical market data without direct intervention in volatile environments.
3. **Robotics and Automation:** By assessing pre-determined paths, passive RL helps in quality control and improving precision in manufacturing where exploration is costly.

Direct Utility Estimation

1. Introduction

Direct Utility Estimation is a learning method used in Passive Reinforcement Learning. In this approach, the agent is given a fixed policy and its task is not to improve the policy, but only to evaluate how good each state is while following that policy.

The agent learns by experience. It observes what rewards it receives when it passes through different states and uses these observations to compute the utility (value) of each state.

2. What is Utility?

The utility of a state is a numerical value that represents how desirable that state is in the long run.

- High utility → good state (leads to high rewards)
- Low utility → bad state (leads to penalties)

Utility considers future rewards, not just immediate rewards.

3. Why Direct Utility Estimation is Needed

In many environments:

- Transition probabilities are unknown
- Reward structure is uncertain
- Building a full model of the environment is difficult

Direct Utility Estimation allows the agent to:

- Learn state utilities without knowing the model
- Use only actual experience

4. Key Assumptions

1. The policy is fixed

   The agent always follows the same action in a given state.
2. The environment is episodic

   Each run ends in a terminal state.
3. Rewards are observable

   The agent can record rewards after each action.

5. How Direct Utility Estimation Works

Step-by-Step Process

1. **Start in an initial state**

   The agent begins an episode.

2. **Follow the fixed policy**

   At each state, the agent takes the action prescribed by the policy.

3. **Reach a terminal state**

   The episode ends when a goal or failure state is reached.

4. **Record rewards**

   The agent records all rewards received during the episode.

5. **Compute return for each state**

   For every state visited, calculate the total reward obtained from that state onward.

6. **Average over multiple episodes**

   After many episodes, the utility of a state is computed as the average of all observed returns from that state.

6. Mathematical Representation

Let:

- $U(s)$= utility of state s
- $N(s)$= number of times state s is visited
- $R_i(s)$= total reward obtained after the i-th visit to state s

$$U(s) = \frac{1}{N(s)} \sum_{i=1}^{N(s)} R_i(s)$$

This formula shows that utility is simply the mean of observed returns.

7. Example Explanation

Example Scenario: Maze Navigation

- A robot follows a fixed path through a maze.
- Each move has a small negative reward (−1).
- Reaching the goal gives a positive reward (+10).

If a state is visited 3 times and the total rewards obtained after that state are:

- Episode 1: +5

- Episode 2: +7
- Episode 3: +6

Then:

$$U(s) = \frac{5 + 7 + 6}{3} = 6$$

This means the state has a utility of 6, indicating it usually leads to good outcomes.

8. Important Characteristics

- Model-free: No transition probabilities required
- Passive learning: No policy improvement
- Experience-based: Learns only from observed data
- Episode-dependent: Updates after full episodes

9. Advantages

1. **Simplicity**

   Easy to understand and implement.

2. **No model required**

   Works without knowing how the environment behaves.

3. **Accurate in the long run**

   With enough episodes, utility values converge to true values.

10. Disadvantages

1. **Slow learning**

   Requires many complete episodes.

2. **Inefficient**

   It does not reuse experience effectively.

3. **Not suitable for large environments**

   High memory and time requirements.

4. **No policy optimization**

   Cannot find better actions.

Adaptive Dynamic Programming (ADP)

1. Introduction

Adaptive Dynamic Programming (ADP) is a reinforcement learning technique used mainly in passive reinforcement learning. In ADP, the agent learns a model of the environment and then uses dynamic programming methods to compute the utility of states.

Unlike Direct Utility Estimation, ADP does not wait for complete episodes to finish. Instead, it updates its estimates continuously using the learned model of the environment, making learning faster and more efficient.

2. Meaning of Adaptive Dynamic Programming

- **Adaptive** → The agent improves its knowledge as it gains experience
- **Dynamic Programming** → Uses recursive equations (Bellman equations) to compute utilities

Definition:

Adaptive Dynamic Programming is a method in which an agent learns the transition and reward model of the environment and applies dynamic programming to estimate the utility of each state under a fixed policy.

3. Role of ADP in Passive Reinforcement Learning

In passive reinforcement learning:

- The policy is fixed
- The agent does not choose actions freely
- The goal is to evaluate how good each state is

ADP helps in:

- Learning the environment model
- Efficiently computing utilities of states
- Faster convergence than simple averaging methods

4. Environment Model in ADP

ADP builds a model of the environment using experience.

4.1 Transition Model

$$P(s' \mid s, a)$$

This represents the probability of moving from state s to state s' after taking action a.

4.2 Reward Model

$$R(s)$$

This represents the immediate reward received when the agent is in state s.

The agent updates these models by observing:

- Current state
- Action taken (from fixed policy)
- Next state
- Reward received

5. Bellman Equation in ADP

Once the environment model is learned, ADP uses the Bellman equation to compute utilities.

$$U(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, a) U(s')$$

Where:

- $U(s)$= Utility of state s
- $R(s)$= Reward of state s
- γ= Discount factor (0 ≤ γ ≤ 1)
- $P(s' \mid s, a)$= Transition probability
- s'= Next state

This equation is applied iteratively until the utility values converge.

6. Working of Adaptive Dynamic Programming

Step-by-Step Process:

1. The agent starts with a fixed policy
2. The agent observes transitions while following the policy
3. It estimates:
    - Transition probabilities
    - Reward values

4. The learned model is updated continuously
5. Dynamic programming is applied using Bellman equations
6. Utility values are refined until convergence

7. Algorithm Outline (Conceptual)

1. Initialize utility values randomly
2. Initialize empty transition and reward models
3. Repeat for each experience:
    - Observe state, action, reward, next state
    - Update model parameters
    - Apply Bellman update to all states
4. Stop when utility values stabilize

8. Advantages of Adaptive Dynamic Programming

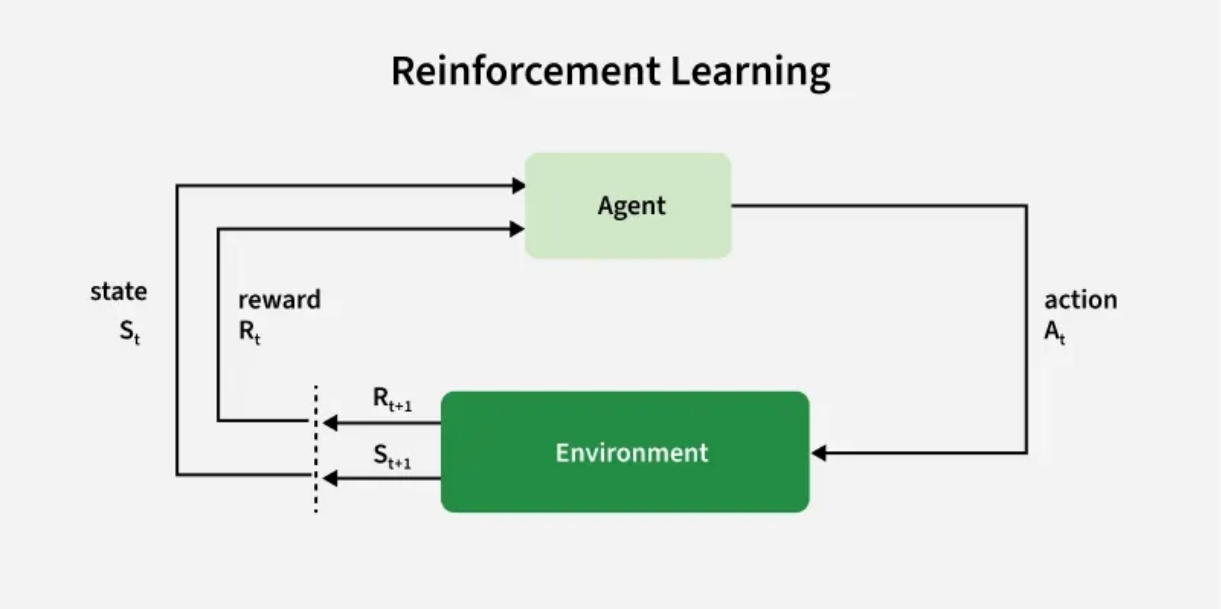
- Faster learning than Direct Utility Estimation
- Efficient use of experience
- Accurate utility estimation
- Suitable for stochastic environments
- Uses full knowledge gained so far

9. Disadvantages of Adaptive Dynamic Programming

- Requires storage of transition models
- Computationally expensive
- Difficult to implement for large state spaces
- Needs accurate probability estimation

Temporal Difference (TD) learning

Temporal Difference (TD) Learning is a model-free reinforcement learning method used by algorithms like Q-learning to iteratively learn state value functions (V(s)) or state-action value functions (Q(s,a)). Unlike Monte Carlo methods that require full episodes, TD Learning updates value estimates after each time step using the Bellman equation and Temporal Difference error.



How TD Learning Combines Monte Carlo and Dynamic Programming

- **Monte Carlo Methods:** Learn value estimates by averaging returns after complete episodes. They require the episode to finish before updating values, which makes them unsuitable for continuing (non-episodic) tasks and can lead to high variance in updates.
- **Dynamic Programming (DP):** Uses a known model of the environment to perform updates by bootstrapping updating value estimates based on other learned values, not just actual returns. DP requires full knowledge of transition and reward probabilities, which is often impractical.

TD Learning: It uses both approaches

- Like Monte Carlo, TD learning does not need a model of the environment (model-free).
- Like DP, TD learning updates predictions based on other learned predictions (bootstrapping), not just actual returns.
- TD methods can learn after every step, not just at episode end, making them suitable for non-episodic and sequential tasks.

Why TD is Model-Free and Non-Episodic

- **Model-Free:** TD learning does not require knowledge of the environment's transition or reward structure. It learns directly from experience by updating values based on observed rewards and subsequent predictions.
- **Non-Episodic (Sequential):** TD learning can update values after every step, not just after an episode ends. This makes it suitable for tasks that do not have clear episode boundaries, such as real-time control or ongoing games.

Core Components

1. Bellman Equation: The foundation for TD updates. For a state value function:

$$V(s) = \mathbb{E}[R_{t+1} + \gamma V(s_{t+1}) \mid s_t = s]$$

And for Action-Value Function (Q-function)

$$Q(s,a) = \mathbb{E}_{\pi}[R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) \mid s_t = s, a_t = a]$$

2. Temporal Difference (TD) Error: $\delta_t = R_{t+1} + \gamma V(s_{t+1}) - V(s_t)$

3. TD Learning Update Rule: $V(s_t) \leftarrow V(s_t) + \alpha\delta_t$

4. Expanded TD Update (in one step): The most basic TD algorithm is TD(0), which updates the value function $V(s)$ for state s as follows:

$$V(s_t) \leftarrow V(s_t) + \alpha[R_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

Where:

- $V(s_t)$: Current estimate for state stst
- R_{t+1}: Reward received after taking action in stst
- γ: Discount factor (how much future rewards are valued)
- $V(s_{t+1})$: Estimated value of the next state
- α: Learning rate

The term $[R_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$ is called the TD error (δ_t), representing the difference between the predicted and the newly observed value.

How TD Learning Works: Example with Bellman Equation

Scenario: Assume a grid world with nine squares:

- One "goal" square (+10 reward),
- One "poison" square (–10 reward),
- All other squares (–1 reward per step).

Suppose the agent starts in state S1. The value of each state, V(s), is initialized to zero.

Step 1: Agent Takes an Action

- The agent in S1 randomly chooses to move right to S2.
- It receives a reward $R_{t+1} = -1$

Step 2: Apply the Bellman Equation

The Bellman equation for the value of a state is:

$$V(s_t) = \mathbb{E}[R_{t+1} + \gamma V(s_{t+1})]$$

Suppose $\gamma = 0$ (for simplicity in this step), and $V(S_2) = 0$

$$V_{\text{obs}}(S_1) = R_{t+1} + \gamma V(S_2) = -1 + 0 = -1$$

Step 3: Calculate Temporal Difference Error

$$\delta t = Vobs(S1) - V(S1) = -1 - 0 = -1$$

Step 4: Update the Value Function

With learning rate $\alpha = 0.1$

$$V(S_1) \leftarrow V(S_1) + \alpha\delta_t = 0 + 0.1 \times (-1) = -0.1$$

Multiple Episodes and Iterative Updates

- **Next time step:** The agent is now in S2, takes an action (say, moves down to S7), and receives a reward ($R_{t+1} = -1$), and the process repeats.
- **After each action:** The value of the current state is updated using the observed reward and the estimated value of the next state.
- **Multiple episodes:** This sequence continues for many episodes. Over time, the values in the table converge to reflect the expected total reward from each state under the current policy.
- **Policy improvement:** Once the value function is learned, the agent can use it to choose actions that maximize expected rewards, shifting from random actions to optimal ones.

This code simulates TD updates for a short sequence. At each step, the value of the current state is updated based on the reward and the value of the next state.

```
import numpy as np

# Example
states = ['A', 'B', 'C', 'D', 'E']
V = {s: 0.0 for s in states}  # Initialize value function
alpha = 0.1   # Learning rate
gamma = 0.9   # Discount factor

# Simulated experience: (current_state, reward, next_state)
```

```
experience = [
   ('B', 0, 'C'),
   ('C', 0, 'D'),
   ('D', 1, 'E'),  # Terminal state E with reward 1
]


for s, r, s_next in experience:
   V[s] = V[s] + alpha * (r + gamma * V[s_next] - V[s])

print(V)
```

Output

{'A': 0.0, 'B': 0.0, 'C': 0.0, 'D': 0.1, 'E': 0.0}

Active Reinforcement Learning

Active Reinforcement Learning is a type of reinforcement learning in which an agent actively selects actions in order to learn the best possible behavior. Unlike passive reinforcement learning, the agent is not provided with a fixed policy. Instead, it must learn the optimal policy by interacting with the environment.

The agent improves its performance over time by receiving rewards or penalties for the actions it performs.

Difference Between Passive and Active Reinforcement Learning

Aspect	Passive Reinforcement Learning	Active Reinforcement Learning
Policy	Fixed	Learned by the agent
Action choice	Not allowed	Allowed
Exploration	No	Yes
Goal	Evaluate state utilities	Find optimal policy

Key Components of Active Reinforcement Learning

- **Agent** – Learns from experience and makes decisions
- **Environment** – Responds to the agent's actions
- **State (s)** – Represents the current situation

- **Action (a)** – Decision taken by the agent
- **Reward (r)** – Feedback from the environment
- **Policy (π)** – Strategy learned by the agent

Exploration vs Exploitation

A major challenge in active reinforcement learning is balancing exploration and exploitation.

Exploration involves trying new or unknown actions to discover better rewards. This may produce low rewards initially but helps the agent learn more about the environment.

Exploitation involves choosing actions that are already known to give high rewards. This maximizes immediate benefit but may prevent the agent from finding better long-term solutions.

An effective learning agent must maintain a balance between both.

Methods Used in Active Reinforcement Learning

Commonly used methods include:

- Q-Learning
- SARSA
- ε-greedy action selection
- Softmax action selection

Q-Learning in Active Reinforcement Learning

Q-Learning is the most widely used algorithm in active reinforcement learning. It is a model-free learning method that allows the agent to learn the optimal policy without knowing the environment model.

The Q-value represents the expected future reward of taking a particular action in a given state.

$$Q(s,a) = \text{Expected future reward of taking action } a \text{ in state } s$$

The Q-Learning update rule is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max Q(s',a') - Q(s,a)]$$

Where:

- α is the learning rate
- γ is the discount factor

- r is the immediate reward
- s' is the next state

Learning Process in Active Reinforcement Learning

The agent observes the current state and selects an action using exploration or exploitation. After performing the action, it receives a reward and moves to the next state. The agent then updates its knowledge (Q-values) and repeats this process until the optimal policy is learned.

Advantages of Active Reinforcement Learning

- Learns optimal behavior automatically
- Adapts to unknown and dynamic environments
- Does not require a predefined policy
- Widely applicable in real-world problems

Disadvantages of Active Reinforcement Learning

- Requires many interactions with the environment
- Slow convergence in large state spaces
- High computational and memory cost
- Initial exploration may reduce performance

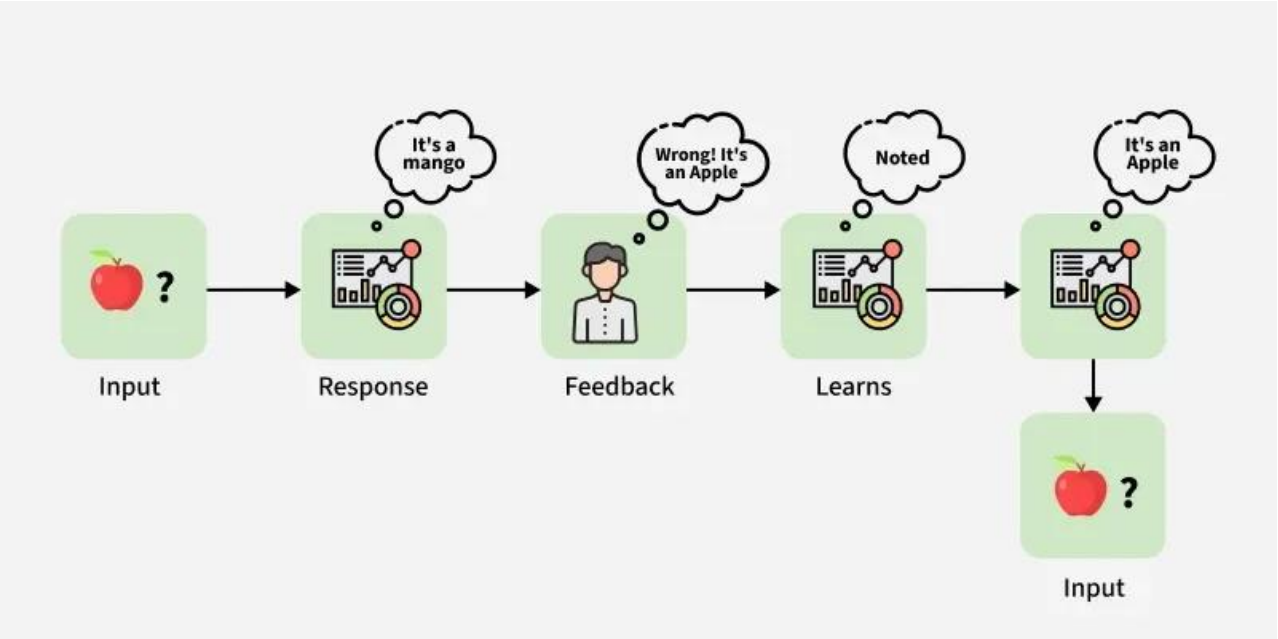
Applications of Active Reinforcement Learning

- Game playing such as Chess and Go
- Robotics and autonomous control systems
- Self-driving vehicles
- Recommendation systems
- Resource allocation and scheduling

Q-Learning in Reinforcement Learning

Q-Learning is a popular model-free reinforcement learning algorithm that helps an agent learn how to make the best decisions by interacting with its environment. Instead of needing a model of the environment the agent learns purely from experience by trying different actions and seeing their results

Imagine a system that sees an apple but incorrectly says, “It’s a mango.” The system is told, “Wrong! It’s an apple.” It learns from this mistake. Next time, when shown the apple, it correctly says “It’s an apple.” This trial-and-error process, guided by feedback is like how Q-Learning works.



Q-Learning

The core idea is that the agent builds a Q-table which stores Q-values. Each Q-value estimates how good it is to take a specific action in a given state in terms of the expected future rewards. Over time the agent updates this table using the feedback it receives

Key Components

1. Q-Values or Action-Values

Q-values represent the expected rewards for taking an action in a specific state. These values are updated over time using the Temporal Difference (TD) update rule.

2. Rewards and Episodes

The agent moves through different states by taking actions and receiving rewards. The process continues until the agent reaches a terminal state which ends the episode.

3. Temporal Difference or TD-Update

The agent updates Q-values using the formula:

$$Q(S,A) \leftarrow Q(S,A) + \alpha(R + \gamma Q(S',A') - Q(S,A))$$

Where,

- S is the current state.

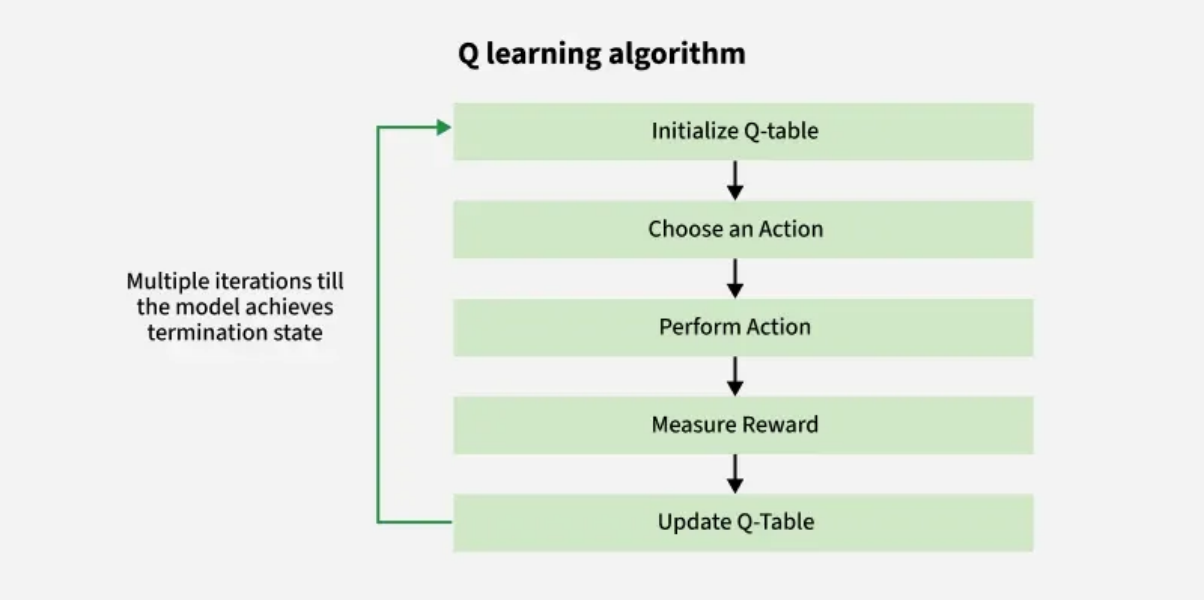
- A is the action taken by the agent.
- S' is the next state the agent moves to.
- A' is the best next action in state S'.
- R is the reward received for taking action A in state S.
- γ (Gamma) is the discount factor which balances immediate rewards with future rewards.
- α (Alpha) is the learning rate determining how much new information affects the old Q-values.

4. ϵ-greedy Policy (Exploration vs. Exploitation)

The ϵ-greedy policy helps the agent decide which action to take based on the current Q-value estimates:

- **Exploitation:** The agent picks the action with the highest Q-value with probability $1 - \epsilon$. This means the agent uses its current knowledge to maximize rewards.
- **Exploration:** With probability ϵ, the agent picks a random action, exploring new possibilities to learn if there are better ways to get rewards. This allows the agent to discover new strategies and improve its decision-making over time.

How does Q-Learning Works?



Q-learning models follow an iterative process where different components work together to train the agent. Here's how it works step-by-step:

Q learning algorithm

1. Start at a State (S)

The environment provides the agent with a starting state which describes the current situation or condition.

2. Agent Selects an Action (A)

Based on the current state and the agent chooses an action using its policy. This decision is guided by a Q-table which estimates the potential rewards for different state-action pairs. The agent typically uses an ε-greedy strategy:

- It sometimes explores new actions (random choice).
- It mostly exploits known good actions (based on current Q-values).

3. Action is Executed and Environment Responds

The agent performs the selected action. The environment then provides:

- A new state (S') — the result of the action.
- A reward (R) — feedback on the action's effectiveness.

4. Learning Algorithm Updates the Q-Table

The agent updates the Q-table using the new experience:

- It adjusts the value for the state-action pair based on the received reward and the new state.
- This helps the agent better estimate which actions are more beneficial over time.

5. Policy is Refined and the Cycle Repeats

With updated Q-values the agent:

- Improves its policy to make better future decisions.
- Continues this loop — observing states, taking actions, receiving rewards and updating Q-values across many episodes.

Over time the agent learns the optimal policy that consistently yields the highest possible reward in the environment.

Methods for Determining Q-values

1. Temporal Difference (TD):

Temporal Difference is calculated by comparing the current state and action values with the previous ones. It provides a way to learn directly from experience, without needing a model of the environment.

2. Bellman's Equation:

Bellman's Equation is a recursive formula used to calculate the value of a given state and determine the optimal action. It is fundamental in the context of Q-learning and is expressed as:

$$Q(s,a) = R(s,a) + \gamma \max_a Q(s',a)$$

Where:

- Q(s, a) is the Q-value for a given state-action pair.
- R(s, a) is the immediate reward for taking action a in state s.
- γ is the discount factor, representing the importance of future rewards.
- $max_a Q(s',a)$ is the maximum Q-value for the next state s' and all possible actions.

UNIT - V

Parallel Artificial Intelligence

Parallel artificial intelligence, at its core, involves dividing a single AI task into smaller subtasks that can be executed simultaneously on multiple processors or cores within a single machine. This approach leverages the power of multi-core processors or specialized hardware like GPUs (Graphics Processing Units) to achieve faster processing times.

Key Characteristics of Parallel AI

- **Shared Memory:** Parallel AI systems typically operate within a shared memory architecture. This means that all processors have direct access to the same memory space, allowing for efficient data sharing and communication.
- **Tight Coupling:** The processors in a parallel system are tightly coupled, meaning they are closely connected and can communicate with each other very quickly.
- **Single Machine Focus:** Parallel AI is primarily focused on leveraging the resources available within a single machine, whether it's a powerful workstation or a dedicated server.

Examples of Parallel AI Implementation

- **Multi-core CPU Training:** Training a neural network on a CPU with multiple cores, where each core handles a portion of the training data or computations.
- **GPU Acceleration:** Using a GPU with thousands of cores to accelerate matrix operations, which are fundamental to many AI algorithms, especially deep learning.
- **Parallel Data Processing:** Splitting a large dataset into smaller chunks and processing each chunk simultaneously on different cores.

Understanding Distributed Artificial Intelligence

Distributed artificial intelligence, in contrast, involves distributing an AI task across multiple machines or nodes connected over a network. Each machine works independently on its assigned portion of the task, and the results are then aggregated to produce the final output.

Key Characteristics of Distributed AI

- **Distributed Memory:** Distributed AI systems operate with distributed memory. Each machine has its own private memory space, and data must be explicitly transferred between machines.
- **Loose Coupling:** The machines in a distributed system are loosely coupled, meaning they are connected over a network and communication can be slower compared to parallel systems.

- **Multiple Machine Focus:** Distributed AI is designed to leverage the combined resources of multiple machines, allowing for the tackling of significantly larger and more complex AI problems.

Examples of Distributed AI Implementation

- **Cluster Computing for Training:** Using a cluster of computers to train a large-scale deep learning model, where each machine processes a subset of the training data and shares its updates with the other machines.
- **Federated Learning:** Training AI models across multiple devices (e.g., smartphones) without sharing the raw data. Each device trains the model locally and sends updates to a central server.
- **Distributed Data Processing with Spark:** Using Apache Spark to process large datasets across a cluster of machines, where each machine performs data transformations and computations in parallel.

Parallel vs. Distributed AI: A Head-to-Head Comparison

To understand which approach is faster, let's compare parallel and distributed AI across several key dimensions:

Feature	Parallel AI	Distributed AI
Architecture	Shared memory, tight coupling	Distributed memory, loose coupling
Communication	Fast, direct access to shared memory	Slower, network-based communication
Scalability	Limited by the resources of a single machine	Highly scalable across multiple machines
Complexity	Simpler to implement and manage	More complex to implement and manage
Cost	Lower initial cost	Higher initial cost, potential for cost savings at scale
Typical Use Cases	Image recognition, natural language processing on moderate datasets	Large-scale data processing, training massive models, federated learning

Psychological Modelling

Introduction

Psychological modelling is an important concept in Artificial Intelligence that focuses on understanding and simulating human cognitive behavior. It attempts to represent how humans think, reason, learn, remember, and make decisions. The goal of psychological modelling is to design AI systems that behave in a human-like manner.

In Parallel and Distributed AI, psychological modelling assumes that human intelligence is not sequential but involves many mental processes operating simultaneously. Hence, AI systems are designed using parallel and distributed structures to imitate human cognition.

Importance of Psychological Modelling in AI

Psychological modelling plays a vital role in AI because it:

- Helps in understanding human intelligence
- Enables the design of intelligent, human-like systems
- Improves human–computer interaction
- Supports the development of cognitive architectures

By mimicking human reasoning, AI systems become more natural, adaptable, and efficient.

Psychological Modelling and Parallelism

Human thinking involves multiple processes occurring at the same time, such as perception, memory recall, and reasoning. Psychological modelling reflects this by using parallel processing.

Examples of parallel cognitive processes:

- Visual perception and decision-making
- Language understanding and memory retrieval
- Problem solving using multiple strategies

Parallel AI systems are therefore suitable for psychological modelling because they can process multiple tasks simultaneously, similar to the human brain.

Psychological Modelling and Distributed AI

In distributed AI, intelligence is spread across multiple agents. Psychological modelling supports this idea by viewing cognition as a result of interacting mental components rather than a single process.

Each agent can represent:

- A specific cognitive function
- A knowledge source
- A reasoning module

The interaction among these agents results in intelligent behaviour.

Cognitive Architectures in Psychological Modelling

Psychological modelling is often implemented using cognitive architectures, which define the structure of the human mind.

Examples:

- ACT-R (Adaptive Control of Thought-Rational)
- SOAR
- CLARION

These architectures:

- Use parallel processing
- Represent memory, learning, and reasoning
- Simulate human problem-solving behavior

Learning in Psychological Modelling

Learning is a key aspect of psychological modelling. AI systems learn by:

- Reinforcing successful actions
- Updating knowledge structures
- Adapting to new situations

This resembles human learning through experience and feedback.

Applications of Psychological Modelling

Psychological modelling is widely used in:

- Intelligent tutoring systems
- Cognitive robots
- Human-computer interaction systems
- Virtual assistants

- Behavioural simulation

These applications require AI systems to understand and predict human behaviour.

Advantages of Psychological Modelling

- Produces human-like intelligent systems
- Improves interaction between humans and machines
- Supports parallel and distributed processing
- Helps in understanding cognition scientifically

Limitations of Psychological Modelling

- Human cognition is complex and difficult to model completely
- Requires interdisciplinary knowledge
- Computationally expensive
- Models may oversimplify human behavior

Parallelism in Reasoning Systems

Introduction

Parallelism in reasoning systems refers to performing multiple reasoning operations at the same time using more than one processor or reasoning unit. Instead of executing inference steps sequentially, the system divides the reasoning task and processes several parts simultaneously, which improves speed, efficiency, and scalability.

Parallel reasoning is inspired by human cognition, where several mental activities occur concurrently.

Need for Parallelism in Reasoning

Parallel reasoning is required when:

- Knowledge bases are very large
- Search spaces grow exponentially
- Real-time reasoning is needed
- Complex problems must be solved quickly

It helps reduce computation time and improves overall system performance.

Types of Parallelism in Reasoning Systems

Data Parallelism

Data parallelism involves applying the same reasoning operation to multiple pieces of data at the same time. Each processor works on a different subset of data but performs the same task.

Explanation:

If a knowledge base contains many facts, the system can check or match rules against multiple facts simultaneously.

Example:
Parallel pattern matching in rule-based systems.

Advantage:
Efficient handling of large datasets.

Task Parallelism

Task parallelism refers to executing different reasoning tasks simultaneously. Each processor handles a separate task or function of the reasoning system.

Explanation:
Reasoning systems consist of multiple tasks such as rule matching, inference, conflict resolution, and explanation generation. These tasks can be executed in parallel.

Example:
One processor applies inference rules while another updates the knowledge base.

Advantage:
Reduces idle time and increases system throughput.

Search Parallelism

Search parallelism involves exploring multiple branches of a search tree at the same time.

Explanation:
In many AI problems, reasoning requires searching through a large solution space. Parallel search allows different processors to explore different paths concurrently.

Example:
Parallel game tree search in chess or planning systems.

Advantage:
Speeds up problem solving in complex search spaces.

Functional Parallelism

Functional parallelism occurs when different functional components of a reasoning system operate concurrently.

Explanation:
Components such as perception, reasoning, learning, and action execution run in parallel rather than sequentially.

Example:
A robot simultaneously perceives the environment, reasons about actions, and executes movements.

Advantage:
Supports real-time intelligent behaviour.

Control Parallelism

Control parallelism refers to the simultaneous execution of multiple control strategies or inference paths.

Explanation:
The system uses multiple reasoning strategies in parallel and selects the best result.

Example:
Using heuristic reasoning and logical reasoning simultaneously.

Advantage:
Improves robustness and flexibility of reasoning.

Parallelism in Rule-Based Reasoning Systems

In rule-based systems:

- Multiple rules may be applicable at the same time
- Parallel rule firing is allowed
- Conflict resolution strategies manage simultaneous execution

This improves inference speed in expert systems.

Parallelism in Logic and Theorem Proving

Parallel reasoning is used in:

- Parallel resolution
- Distributed theorem proving
- Concurrent constraint satisfaction

Multiple logical inferences are performed simultaneously to reach conclusions faster.

Advantages of Parallelism in Reasoning Systems

- Faster reasoning and inference
- Efficient handling of complex problems
- Scalability for large knowledge bases
- Better real-time performance

Limitations of Parallelism in Reasoning Systems

- Increased system complexity
- Communication and synchronization overhead
- Difficult debugging and maintenance
- Hardware dependency

Applications of Parallel Reasoning Systems

- Expert systems
- Automated planning
- Game playing systems
- Robotics
- Natural language processing

Distributed Reasoning Systems

Introduction

Distributed reasoning systems are AI systems in which reasoning is performed by multiple autonomous agents rather than a single centralized system. Each agent has its own knowledge, reasoning capability, and decision-making power. These agents communicate and cooperate to solve complex problems.

Distributed reasoning is especially useful when problems are large, complex, or spread across different locations.

Concept of Distributed Reasoning

In distributed reasoning systems:

- Knowledge is distributed among agents
- Reasoning is performed locally by each agent

- Agents share results through communication
- A global solution emerges from cooperation

This approach reflects human collaborative problem solving, where individuals work together to reach a decision.

Components of Distributed Reasoning Systems

- **Agents** – Autonomous entities capable of reasoning
- **Local Knowledge Base** – Knowledge specific to each agent
- **Inference Engine** – Performs reasoning locally
- **Communication Mechanism** – Enables message passing
- **Coordination Strategy** – Manages cooperation among agents

Working of Distributed Reasoning Systems

Each agent observes its environment and reasons using its local knowledge. When necessary, agents communicate partial results or requests to other agents. The agents then coordinate and combine their reasoning outcomes to achieve a common goal.

Types of Distributed Reasoning Systems

Cooperative Distributed Reasoning

All agents work toward a common goal and share information freely.

Example:
Multi-robot systems performing a task together.

Advantage:
Efficient and reliable problem solving.

Competitive Distributed Reasoning

Agents have individual goals and may compete for resources.

Example:
Automated trading agents.

Advantage:
Encourages optimal strategies.

Hybrid Distributed Reasoning

Combines both cooperation and competition among agents.

Example:
Negotiation systems.

Communication in Distributed Reasoning

Agents communicate using:

- Message passing
- Shared knowledge repositories
- Blackboard systems

Communication is essential for coordination and conflict resolution.

Advantages of Distributed Reasoning Systems

- Scalability to large systems
- Fault tolerance
- Efficient use of resources
- Parallel problem solving

Disadvantages of Distributed Reasoning Systems

- Communication overhead
- Synchronization issues
- Complex coordination
- Difficult debugging

Applications of Distributed Reasoning Systems

- Multi-agent systems
- Distributed expert systems
- Sensor networks
- Robotics and automation
- Intelligent transportation systems

Planning

Introduction

Planning in Artificial Intelligence is the process of deciding a sequence of actions that an agent must perform to achieve a specific goal from a given initial state. Planning enables intelligent systems to act purposefully by considering future consequences of actions.

In parallel and distributed AI, planning becomes more complex because multiple agents may be involved, each with its own goals, actions, and resources.

Meaning of Planning

Planning is the ability of an AI system to:

- Analyze the current state
- Define goals
- Select appropriate actions
- Arrange actions in a logical order

The outcome of planning is a plan, which is a sequence of actions that leads to the goal.

Components of a Planning System

- **Initial State** – The starting situation of the system
- **Goal State** – The desired outcome
- **Actions / Operators** – Possible actions the agent can perform
- **Preconditions** – Conditions that must be true before an action
- **Effects** – Changes produced by an action

Classical Planning

Classical planning assumes:

- A single agent
- A static and deterministic environment
- Complete knowledge of the world

Examples include:

- STRIPS planning
- State-space planning

Planning in Parallel and Distributed AI

In parallel and distributed AI:

- Planning is done by multiple agents
- Each agent plans for a part of the problem
- Agents must coordinate and communicate
- Conflicts and dependencies must be handled

This is known as distributed planning.

Distributed Planning

Distributed planning is a planning approach where:

- Multiple agents create plans independently
- Partial plans are coordinated to form a global plan
- Agents share information and resources

It is useful when centralized planning is not feasible.

Parallel Planning

Parallel planning allows:

- Multiple actions to be executed at the same time
- Faster plan execution
- Efficient use of resources

Actions can be executed in parallel if they do not conflict with each other.

Coordination in Distributed Planning

Coordination is required to:

- Avoid conflicts
- Share resources
- Synchronize actions

Techniques include:

- Negotiation
- Task allocation
- Communication protocols

Planning Algorithms in Distributed AI

Some common approaches:

- Multi-agent planning
- Partial-order planning
- Hierarchical planning
- Blackboard-based planning

Advantages of Planning in Distributed AI

- Scalability
- Faster plan generation
- Robustness
- Efficient problem solving

Disadvantages of Planning in Distributed AI

- Communication overhead
- Complexity in coordination
- Increased computation
- Difficult conflict resolution

Applications of Planning

- Robotics and automation
- Logistics and supply chain management
- Autonomous vehicles
- Smart manufacturing
- Multi-robot coordination

Fault Tolerance

Introduction

Fault tolerance refers to the ability of a system to continue functioning correctly even when some of its components fail. In parallel and distributed AI systems, failures are common due to the presence of multiple processors, agents, and communication links. Fault tolerance ensures that such failures do not stop the entire system.

Fault-tolerant systems are essential for reliable, large-scale, and mission-critical AI applications.

Meaning of Fault Tolerance

Fault tolerance is the capability of an AI system to:

- Detect faults
- Isolate faulty components
- Recover from failures
- Continue operation with minimal performance loss

The system adapts itself automatically without human intervention.

Types of Faults in Distributed AI Systems

- Hardware faults occur due to processor or memory failures.
- Software faults arise from bugs or incorrect logic.
- Communication faults happen when messages are lost or delayed.
- Agent faults occur when an intelligent agent stops responding or behaves incorrectly.

Need for Fault Tolerance in Parallel and Distributed AI

Fault tolerance is necessary because:

- Components are geographically distributed
- Systems operate continuously
- Failures are unavoidable
- Human intervention is costly

Fault tolerance improves system reliability and availability.

Fault Tolerance Techniques

Redundancy
Multiple components perform the same task. If one fails, another takes over.

Replication
Data or agents are duplicated to ensure availability.

Checkpointing and Recovery

System state is saved periodically and restored after failure.

Graceful Degradation

System continues to operate with reduced performance.

Failure Detection and Isolation

Faulty components are identified and isolated from the system.

Fault Tolerance in Multi-Agent Systems

In multi-agent systems:

- Agents operate independently
- Failure of one agent does not affect others
- Tasks can be reassigned
- Agents cooperate to recover from failures

This makes distributed AI systems naturally fault tolerant.

Advantages of Fault Tolerance

- Increased system reliability
- Continuous operation
- Reduced downtime
- Improved user trust

Limitations of Fault Tolerance

- Increased system complexity
- Higher cost due to redundancy
- Performance overhead
- Difficult implementation

Applications of Fault-Tolerant AI Systems

- Autonomous vehicles
- Space exploration systems
- Healthcare monitoring systems
- Industrial automation
- Cloud-based AI services