Paper/ Subject Code: 53704 / Artificial Intelligence

(NOV 2018) **Q.P. code: 57835**

Q.1 a) What is Artificial Intelligence? State its applications. (5)

- AI is one of the newest fields in science and engineering.
- AI is a general term that implies the use of a computer to model & replicate intelligent behaviour.
- "AI is the design, study & construction of computer programs that behave intelligently."
- Artificial intelligence (AI) refers to the simulation of human intelligence in machines that are programmed to think like humans and mimic their actions. The term may also be applied to any machine that exhibits traits associated with a human mind such as learning and problem-solving.
- The ideal characteristic of artificial intelligence is its ability to rationalize and take actions that have the best chance of achieving a specific goal.
- AI is continuously evolving to benefit many different industries. Machines are wired using a cross-disciplinary approach based in mathematics, computer science, linguistics, psychology, and more.
- Research in AI focuses on development & analysis of algorithms that learn & perform intelligent behaviour with minimal human intervention.
- AI is the ability of machine or computer program to think and learn.
- The concept of AI is based on idea of building machines capable of thinking, acting & learning like humans.
- AI is only field to attempt to build machines that will function autonomously complex changing environments.
- AI has focused chiefly on following components of intelligence.
    - **Learning: -** the learning by trial & error.
    - **Reasoning: -** reasoning skill often happen subconsciously & within seconds.
    - **Decision making: -** it is a process of making choices by identifying a decision gathering information & assessing alternative resolutions.
    - **Problem solving: -** problem solving particularly in AI may be characterized as systematic search in order to reach goal or solutions.

Artificial Intelligence has various applications in today's society. It is becoming essential for today's time because it can solve complex problems with an efficient way in multiple industries, such as Healthcare, entertainment, finance, education, etc. AI is making our daily life more comfortable and fast.

Following are some sectors which have the application of Artificial Intelligence:

I. **AI in Astronomy**

- Artificial Intelligence can be very useful to solve complex universe problems. AI technology can be helpful for understanding the universe such as how it works, origin, etc.

II. **AI in Healthcare**

- In the last, five to ten years, AI becoming more advantageous for the healthcare industry and going to have a significant impact on this industry.
- Healthcare Industries are applying AI to make a better and faster diagnosis than humans. AI can help doctors with diagnoses and can inform when patients are worsening so that medical help can reach to the patient before hospitalization.

III. **AI in Gaming**

- AI can be used for gaming purpose. The AI machines can play strategic games like chess, where the machine needs to think of a large number of possible places.

IV. **AI in Finance**

- AI and finance industries are the best matches for each other. The finance industry is implementing automation, chatbot, adaptive intelligence, algorithm trading, and machine learning into financial processes.

V. **AI in Data Security**

- The security of data is crucial for every company and cyber-attacks are growing very rapidly in the digital world. AI can be used to make your data more safe and secure. Some examples such as AEG bot, AI2 Platform, are used to determine software bug and cyber-attacks in a better way.

VI. **AI in Social Media**

- Social Media sites such as Facebook, Twitter, and Snapchat contain billions of user profiles, which need to be stored and managed in a very efficient way. AI can organize and manage massive amounts of data. AI can analyze lots of data to identify the latest trends, hashtag, and requirement of different users.

VII. **AI in Travel & Transport**

- AI is becoming highly demanding for travel industries. AI is capable of doing various travel related works such as from making travel arrangement to suggesting the hotels, flights, and best routes to the customers. Travel industries are using AI-powered

chatbots which can make human-like interaction with customers for better and fast response.

VIII. AI in Automotive Industry

- Some Automotive industries are using AI to provide virtual assistant to their user for better performance. Such as Tesla has introduced TeslaBot, an intelligent virtual assistant.
- Various Industries are currently working for developing self-driven cars which can make your journey more safe and secure.

IX. AI in Robotics:

- Artificial Intelligence has a remarkable role in Robotics. Usually, general robots are programmed such that they can perform some repetitive task, but with the help of AI, we can create intelligent robots which can perform tasks with their own experiences without pre-programmed.
- Humanoid Robots are best examples for AI in robotics, recently the intelligent Humanoid robot named as Erica and Sophia has been developed which can talk and behave like humans.

X. AI in Entertainment

- We are currently using some AI based applications in our daily life with some entertainment services such as Netflix or Amazon. With the help of ML/AI algorithms, these services show the recommendations for programs or shows.

Q.1 b) Discuss Turing test with Artificial Intelligence approach. (5)

I. The Turing Test, proposed by Alan Turing (1950), was designed to provide a satisfactory operational definition of intelligence. To judge whether the system can act like a human, Sir Alan turing had designed a test known as turing test.

II. A Turing Test is a method of inquiry in artificial intelligence (AI) for determining whether or not a computer is capable of thinking like a human being.

III. A computer passes the test if a human interrogator, after posing some written questions, cannot tell whether the written responses come from a person or from a computer. Programming a computer to pass a rigorously applied test provides plenty to work on. The computer would need to possess the following capabilities:

1. **Natural language processing** to enable it to communicate successfully in English;
2. **Knowledge representation** to store what it knows or hears;

3. **Automated reasoning** to use the stored information to answer questions and to draw new conclusions;
4. **Machine learning** to adapt to new circumstances and to detect and extrapolate patterns.

IV. Turing's test deliberately avoided direct physical interaction between the interrogator and the computer, because physical simulation of a person is unnecessary for intelligence. However, the so-called **total Turing Test** includes a video signal so that the interrogator can test the subject's perceptual abilities, as well as the opportunity for the interrogator to pass physical objects "through the hatch." To pass the total Turing Test, the computer will need

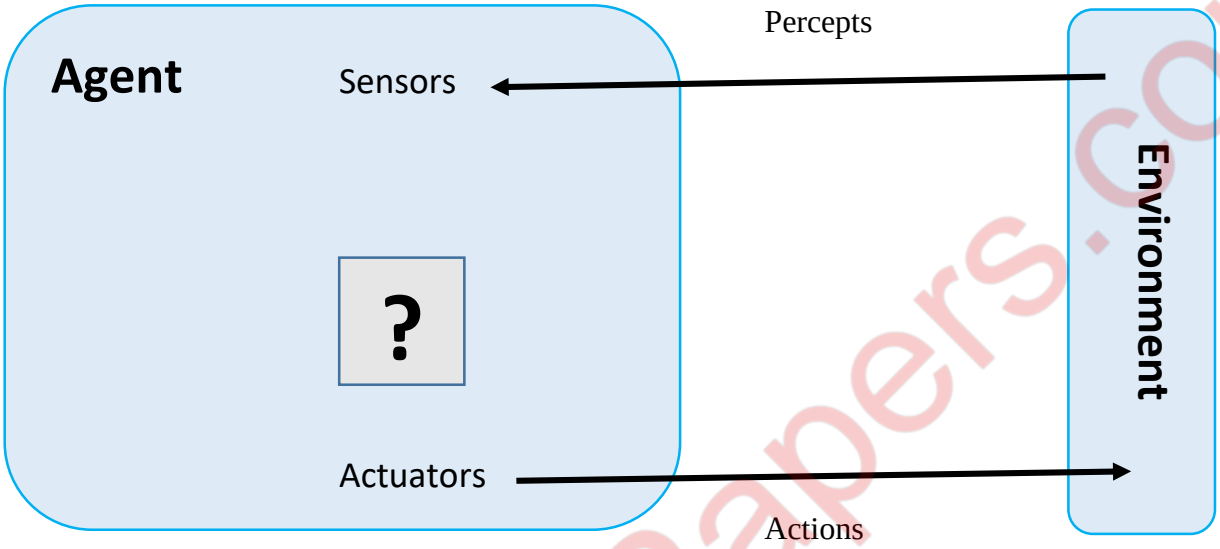
5. **Computer vision** to perceive objects, and
6. **Robotics** to manipulate objects and move about.

V. These six disciplines compose most of AI, and Turing deserves credit for designing a test that remains relevant 60 years later. Yet AI researchers have devoted little effort to passing the Turing Test, believing that it is more important to study the underlying principles of intelligence than to duplicate an exemplar.

Q.1 c) What are agents? Explain how they interact with environment. (5)

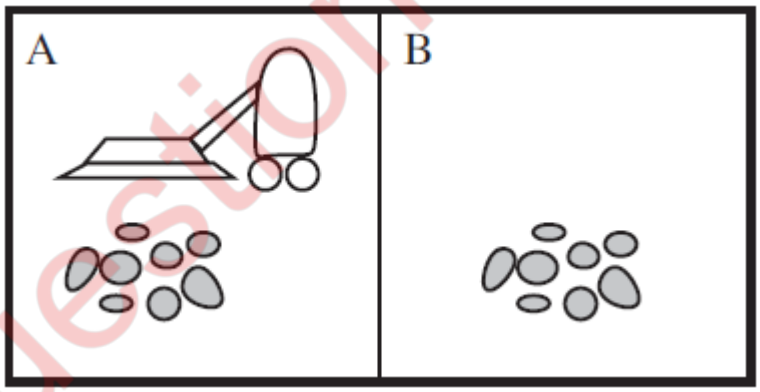
- An **agent** is anything that can be viewed as perceiving its **environment** through **sensors** and acting upon that environment through **actuators**.
- A human agent has eyes, ears, and other organs for sensors and hands, legs, vocal tract, and so on for actuators.
- Eyes, ears, nose, skin, tongue. These senses sense the environment are called as sensors. Sensors collect percepts or inputs from environment and passes it to the processing unit.
- Actuators or effectors are the organs or tools using which the agent acts upon the environment. Once the sensor senses the environment, it gives this information to nervous system which takes appropriate action with the help of actuators. In case of human agents we have hands, legs as actuators or effectors.
- A robotic agent might have cameras and infrared range finders for sensors and various motors for actuators.
- 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.
- Use the term percept to refer to the agent's perceptual inputs at any given instant. An agent's percept sequence is the complete history of everything the agent has ever perceived.
- In general, an agent's choice of action at any given instant can depend on the entire percept sequence observed to date, but not on anything it hasn't perceived.

- By specifying the agent's choice of action for every possible percept sequence, we have said more or less everything there is to say about the agent. Mathematically speaking, we say that an agent's behaviour is described by the agent function that maps any given percept sequence to an action.



Agents interact with environments through sensors and actuators

Take a simple example of vacuum cleaner agent.



- As shown in figure, there are two blocks A & B having some dirt. Vacuum cleaner agent supposed to sense the dirt and collect it, thereby making the room clean.
- In order to do that the agent must have a camera to see the dirt and a mechanism to move forward, backward, left and right to reach to the dirt. Also it should absorb the dirt. Based on the percepts, actions will be performed. For example: Move left, Move right, absorb, No Operation.
- Hence the sensor for vacuum cleaner agent can be camera, dirt sensor and the actuator can be motor to make it move, absorption mechanism. And it can be represented as [A, Dirty], [B, Clean], [A, Absorb], [B, Nop], etc.

Types of Environment

I. Fully observable vs. partially observable:

- If an agent's sensors give it access to the complete state of the environment at each point in time, then we say that the task environment is fully observable.
- Fully observable environments are convenient because the agent need not maintain any internal state to keep track of the world. An environment might be partially observable because of noisy and inaccurate sensors or because parts of the state are simply missing from the sensor data.
- If the agent has no sensors at all then the environment is unobservable.

II. Single agent vs. multiagent:

- An agent solving a crossword puzzle by itself is clearly in a single-agent environment, while in case of car driving agent, there are multiple agents driving on the road, hence it's a multiagent environment.
- For example, in chess, the opponent entity B is trying to maximize its performance measure, which, by the rules of chess, minimizes agent A's performance measure. Thus, chess is a competitive multiagent environment.
- In the taxi-driving environment, on the other hand, avoiding collisions maximizes the performance measure of all agents, so it is a partially cooperative multiagent environment. It is also partially competitive because, for example, only one car can occupy a parking space.

III. Deterministic vs. stochastic:

- If the next state of the environment is completely determined by the current state and the action executed by the agent, then we say the environment is deterministic; otherwise, it is stochastic.
- If the environment is partially observable, however, then it could appear to be stochastic.

IV. Episodic vs. sequential:

- In an episodic task environment, the agent's experience is divided into atomic episodes. In each episode the agent receives a percept and then performs a single action.
- Crucially, the next episode does not depend on the actions taken in previous episodes. Many classification tasks are episodic.
- In sequential environments, on the other hand, the current decision could affect all future decisions.
- Episodic environments are much simpler than sequential environments because the agent does not need to think ahead.

V. Static vs. dynamic:

- If the environment can change while an agent is deliberating, then we say the environment is dynamic for that agent; otherwise, it is static.
- Static environments are easy to deal with because the agent need not keep looking at the world while it is deciding on an action, nor need it worry about the passage of time.
- Dynamic environments, on the other hand, are continuously asking the agent what it wants to do; if it hasn't decided yet, that counts as deciding to do nothing.

- If the environment itself does not change with the passage of time but the agent's performance score does, then we say the environment is semi-dynamic.

VI. Discrete vs. continuous:

- The discrete/continuous distinction applies to the state of the environment, to the way time is handled, and to the percepts and actions of the agent.
- For example, the chess environment has a finite number of distinct states (excluding the clock).
- Chess also has a discrete set of percepts and actions.
- Taxi driving is a continuous-state and continuous-time problem: the speed and location of the taxi and of the other vehicles sweep through a range of continuous values and do so smoothly over time.
- Taxi-driving actions are also continuous (steering angles, etc.). Input from digital cameras is discrete, strictly speaking, but is typically treated as representing continuously varying intensities and locations.

VII. Known vs. unknown:

- In known environment, the output for all probable actions is given. state of knowledge about the "laws of physics" of the environment.
- In case of unknown environment, for an agent to make a decision, it has to gain knowledge about how the environments works.

Q.1 d) What is rational agent? Discuss in brief about rationality. (5)

Rational Agent:

For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, based on the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

1. The concept of rational agents as central to our approach to artificial intelligence.
2. Rationality is distinct from omniscience (all-knowing with infinite knowledge)
3. Agents can perform actions in order to modify future percepts so as to obtain useful information (information gathering, exploration)
4. An agent is autonomous if its behaviour is determined by its own percepts & experience (with ability to learn and adapt) without depending solely on build-in knowledge
5. A rational agent is one that does the right thing—conceptually speaking, every entry in the table for the agent function is filled out correctly. Obviously, doing the right thing is better than doing the wrong thing, but what does it mean to do the right thing?
6. If the sequence is desirable, then the agent has performed well. This notion of desirability is captured by a performance measure that evaluates any given sequence of environment states.
7. For every percept sequence a built-in knowledge base is updated, which is very useful for decision making, because it stores the consequences of performing some particular action.

8. If the consequences direct to achieve desired goal then we get a good performance measure factor, else if the consequences do not lead to desired goal sate, then we get a poor performance measure factor.
   For example :- if agents hurts his finger while using nail and hammer, then while using it for the next time agent will be more careful and the probability of not getting hurts will increase. In short agent will be able to use the hammer and nail more efficiently.
9. A rational agent should be autonomous—it should learn what it can to compensate for partial or incorrect prior knowledge.
10. Rational agent not only to gather information but also to learn as much as possible from what it perceives.
11. After sufficient experience of its environment, the behaviour of a rational agent can become effectively independent of its prior knowledge. Hence, the incorporation of learning allows one to design a single rational agent that will succeed in a vast variety of environments.
12. What is rational at any given time depends on four things:
    - The performance measure that defines the criterion of success.
    - The agent's prior knowledge of the environment.
    - The actions that the agent can perform.
    - The agent's percept sequence to date.

Acting rationally: The rational agent approach

- An **agent** is just something that acts (agent comes from the Latin agere, to do). Of course, all computer programs do something, but computer agents are expected to do more: operate autonomously, perceive their environment, persist over a prolonged time period, and adapt to change, and create and pursue goals.
- A **rational agent** is one that acts so as to achieve the best outcome or, when there is uncertainty, the best expected outcome. In some situations, there is no provably correct thing to do, but something must still be done. There are also ways of acting rationally that cannot be said to involve inference. For example, recoiling from a hot stove is a reflex action that is usually more successful than a slower action taken after careful deliberation.
- All the skills needed for the Turing Test also allow an agent to act rationally. Knowledge representation and reasoning enable agents to reach good decisions. We need to be able to generate comprehensible sentences in natural language to get by in a complex society. We need learning not only for erudition, but also because it improves our ability to generate effective behaviour.
- The rational-agent approach has **two advantages** over the other approaches. First, it is more general than the "laws of thought" approach because correct inference is just one of several possible mechanisms for achieving rationality. Second, it is more amenable to scientific development than are approaches based on human behaviour or human thought. The standard of rationality is mathematically well defined and completely general, and can be "unpacked" to generate agent designs that provably achieve it.

- One important point to keep in mind: We will see before too long that achieving perfect rationality—**always doing the right thing**—is not feasible in complicated environments.

Q.1 e) Explain PEAS description of task environment for automated taxi. (5)

PEAS stands for **P**erformance, **E**nvironment, **A**ctuators, **and Sensors**. It is the short form used for performance issues grouped under task environment.

I. **Performance Measure:**
First, what is the **performance measure** to which we would like our automated driver to aspire? Desirable qualities include getting to the correct destination; minimizing fuel consumption and wear and tear; minimizing the trip time or cost; minimizing violations of traffic laws and disturbances to other drivers; maximizing safety and passenger comfort; maximizing profits.

II. **Environment:**
Next, what is the driving environment that the taxi will face? Any taxi driver must deal with a variety of roads, ranging from rural lanes and urban alleys to 12-lane freeways.
The roads contain other traffic, pedestrians, stray animals, road works, police cars, puddles, and potholes. The taxi must also interact with potential and actual passengers.

III. **Actuators:**
The actuators for an automated taxi include those available to a human driver: control over the engine through the accelerator and control over steering and braking. In addition, it will need output to a display screen or voice synthesizer to talk back to the passengers, and perhaps some way to communicate with other vehicles, politely or otherwise.

IV. **Sensors:**
The basic sensors for the taxi will include one or more controllable video cameras so that it can see the road; it might augment these with infrared or sonar sensors to detect distances to other cars and obstacles. To avoid speeding tickets, the taxi should have a speedometer, and to control the vehicle properly, especially on curves, it should have an accelerometer.

PEAS description of task environment for automated taxi

- **Performance measure:**
  - Safe
  - Fast
  - Optimum speed
  - Legal
  - comfortable trip
  - maximize profits

- **Environment:**
  - Roads
  - other traffic
  - pedestrians
  - customers
- **Actuators:**
  - Steering wheel
  - Accelerator
  - Brake
  - Signal
  - horn
- **Sensors:**
  - Cameras
  - Sonar
  - Speedometer
  - GPS
  - Odometer
  - engine sensors
  - keyboard

Q.1 f) Give comparison between Full observable and partially observable agent. (5)

Sr. No.	Full Observable Agent	Partially Observable Agent
I.	Fully observable environment is one in which the agent can always see the entire state of environment.	Partially observable environment is one in which the agent can never see the entire state of environment.
II.	In case of fully observable environments all relevant portions of the environment are observable.	In case of partially observable environments not all relevant portions of the environment are observable.
III.	Fully observable environment not need memory to make an optimal decision.	Partially observable environment need memory to make an optimal decision.
IV.	A fully observable environment agents are able to gather all the necessary information required to take actions.	A partially observable environment agents cannot provide errorless information at any given time for every internal state, as the environment is not seen completely at any point of time.

V.	In case of fully observable environments agents don't have to keep records of internal states.	In case of partially observable environments agents have to keep records of internal states.
VI.	Examples: - Word block problem, 8-puzzle problem, Sudoku puzzle, cross word puzzle, Checkers with clock etc.	Examples: - Car driving, Part-picking robot, Soccer game.
VII.	Checker Game is the example of full observable environment because the agent has complete knowledge of the board.	Poker game is an example of partially observable environment because the cards are not openly on the table (agent cannot see the opponent hand). So everything about the environment is not accessible.

Q.2 a) Discuss in brief the formulation of single state problem. (5)

I. Problem Formulation

- Goal formulation World states with certain properties
- Definition of the state space (important: only the relevant aspects → abstraction
- Definition of the actions that can change the world state
- Definition of the problem type, which is dependent on the knowledge of the world states and actions → states in the search space
- Determination of the search cost (search costs, offline costs) and the execution costs (path costs, online costs)

II. Single-state problem

- Observable (at least the initial state)
- Deterministic
- Static
- Discrete

III. Single-state problem

- Complete world state knowledge complete action knowledge→The agent always knows its world state

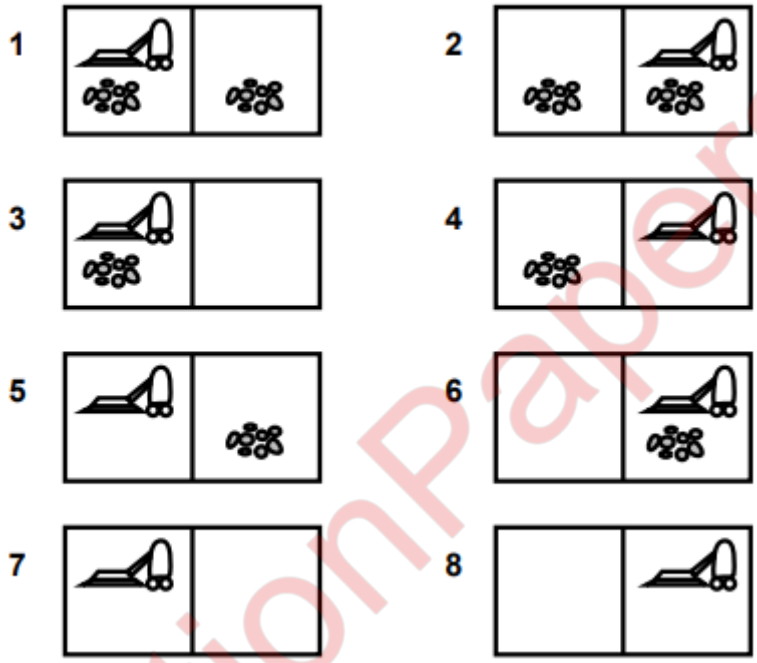
IV. Single state Problem can be defined by 5 components

1. **Initial State:** the state the agent starts
2. **Actions:** the set of operators that can be executed at a state
3. **Transition model:** returns the state that results from doing an action in a state
4. **Goal test:** determines whether a given state is a goal state

5. **Path Cost:** function that assigns a numeric cost to a path

V. The Vacuum Cleaner Problem as a Single-State Problem

- If the environment is completely accessible, the vacuum cleaner always knows where it is and where the dirt is. The solution then is reduced to searching for a path from the initial state to the goal state.
- States for the search: The world states 1-8.



VI. Single-state problem

- exact prediction is possible
- **state -** is known exactly after any sequence of actions
- **accessibility** of the world all essential information can be obtained through sensors
- **consequences** of actions are known to the agent
- **goal** - for each known initial state, there is a unique goal state that is guaranteed to be reachable via an action sequence simplest case, but severely restricted

VII. Vacuum world,

Limitations:

- Can't deal with incomplete accessibility
- incomplete knowledge about consequences changes in the world
- indeterminism in the world, in action

VIII. Example:

Single-state problem formulation

- A problem is defined by 4 items: – initial state e.g., "at Makamba" – operators (or successor function S(x)), e.g., Makamba Mabanda, Makamba Rutana – goal test, can be
- Explicit, e.g., x = "at Bujumbura"
- Implicit, e.g., NoDirt(x) – path cost function, e.g., sum of distances, number of operators executed, etc.
- A solution is a sequence of operators leading from initial state to goal state, e.g., Makamba→ Mabanda→ Bururi→ Bujumbura

IX. Example:

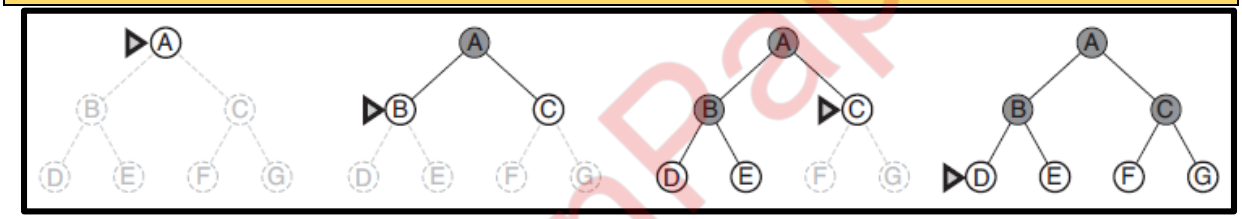
- initial state e.g., "at Arad"
- actions: Actions(s) returns applicabile actions in s: (Go(Sibiu), Go(Timisoara), Go(Zerind)
- transition model - set of action–state pairs (succesor function Result(s,a)): e.g., Result(In(Arad), Go(Zerind))=In(Zerind)
- goal test - determines if whether a given state is a goal state explicit, e.g., xs = "at Bucharest" implicit, e.g., xs = checkmate
- path cost (additive) - reflects agent's own performance measure e.g., sum of distances, number of actions executed, etc. c(s, a, s′) is the step cost, assumed to be ≥ 0

Q.2 b) Give the outline of Breadth First Search algorithm. (5)

- **Breadth-first search** is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.
- Breadth-first search is an instance of the general graph-search algorithm in which the shallowest unexpanded node is chosen for expansion. This is achieved very simply by using a **FIFO queue** for the frontier. Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first.
- There is one slight tweak on the general graph-search algorithm, which is that the goal test is applied to each node when it is generated rather than when it is selected for expansion. This decision is explained below, where we discuss time complexity.
- Note also that the algorithm, following the general template for graph search, discards any new path to a state already in the frontier or explored set; it is easy to see that any such path must be at least as deep as the one already found. Thus, breadth-first search always has the shallowest path to every node on the frontier.
- Pseudocode is given in Figure shows the progress of the search on a simple binary tree.

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ←a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ←a FIFO queue with node as the only element
    explored ←an empty set
    loop do
        if EMPTY?( frontier) then return failure
        node←POP( frontier ) /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
          child ←CHILD-NODE(problem, node, action)
         if child .STATE is not in explored or frontier then
           if problem.GOAL-TEST(child .STATE) then return SOLUTION(child )
           frontier ←INSERT(child , frontier )
```

Breadth-first search on a graph



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

I. We can easily see that it is complete—if the shallowest goal node is at some finite depth d, breadth-first search will eventually find it after generating all shallower nodes (provided the branching factor b is finite). Note that as soon as a goal node is generated, we know it is the shallowest goal node because all shallower nodes must have been generated already and failed the goal test. Now, the shallowest goal node is not necessarily the optimal one; technically, breadth-first search is optimal if the path cost is a non-decreasing function of the depth of the node. The most common such scenario is that all actions have the same cost.

II. So far, the news about breadth-first search has been good. The news about time and space is not so good. Imagine searching a uniform tree where every state has b successors. The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b^2 at the second level. Each of these generates b more nodes, yielding b^3 nodes at the third level, and so on.

III. Now suppose that the solution is at depth d. In the worst case, it is the last node generated at that level. Then the total number of nodes generated is

b + b^2 + b^3 + ▪ ▪ ▪ + b^d = O(b^d) .

(If the algorithm were to apply the goal test to nodes when selected for expansion, rather than when generated, the whole layer of nodes at depth d would be expanded before the goal was detected and the time complexity would be O(b^(d+1)).)

IV. As for space complexity: for any kind of graph search, which stores every expanded node in the explored set, the space complexity is always within a factor of b of the time complexity. For breadth-first graph search in particular, every node generated remains in memory. There will be O(b^(d−1)) nodes in the explored set and O(b^d) nodes in the frontier, so the space complexity is O(b^d), i.e., it is dominated by the size of the frontier.

V. Switching to a tree search would not save much space, and in a state space with many redundant paths, switching could cost a great deal of time. An exponential complexity bound such as O(b^d) is scary.

VI. The memory requirements are a bigger problem for breadth-first search than is the execution time. Fortunately, other strategies require less memory. Time is still a major factor. If your problem has a solution at depth 16, then (given our assumptions) it will take about 350 years for breadth-first search (or indeed any uninformed search) to find it. In general, exponential complexity search problems cannot be solved by uninformed methods for any but the smallest instances.

Q.2 c) Give the outline of tree search algorithm. (5)

I. **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.

II. **Tree** is a hierarchical data structure which stores the information naturally in the form of hierarchy unlike linear data structures like, Linked List, Stack, etc. A tree contains nodes(data) and connections(edges) which should not form a cycle.

III. Following are the few frequently used terminologies for Tree data structure.

- **Node —** A node is a structure which may contain a value or condition, or represent a separate data structure.
- **Root —** The top node in a tree, the prime ancestor.
- **Child —** A node directly connected to another node when moving away from the root, an immediate descendant.
- **Parent —** The converse notion of a child, an immediate ancestor.
- **Leaf —** A node with no children.
- **Internal node —** A node with at least one child.
- **Edge —** The connection between one node and another.
- **Depth —** The distance between a node and the root.
- **Level —** the number of edges between a node and the root + 1

- **Height —** The number of edges on the longest path between a node and a descendant leaf.
- **Breadth —** The number of leaves.
- **Sub Tree —** A tree T is a tree consisting of a node in T and all of its descendants in T.
- **Binary Tree —** is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child.
- **Binary Search Tree** — is a special type of binary tree which has the following properties.

✓ The left subtree of a node contains only nodes with keys lesser than the node's key.
✓ The right subtree of a node contains only nodes with keys greater than the node's key.
✓ The left and right subtree each must also be a binary search tree.

IV. "In computer science, **tree traversal** (also known as **tree search**) is a form of graph traversal and refers to the process of visiting (checking and/or updating) each node in a tree data structure, exactly once. Such traversals are classified by the order in which the nodes are visited."

V. Tree Traversal Algorithms can be classified broadly in the following two categories by the order in which the nodes are visited:

- **Depth-First Search (DFS) Algorithm:** It starts with the root node and first visits all nodes of one branch as deep as possible of the chosen Node and before backtracking, it visits all other branches in a similar fashion. There are three sub-types under this, which we will cover in this article.
- **Breadth-First Search (BFS) Algorithm:** It also starts from the root node and visits all nodes of current depth before moving to the next depth in the tree. We will cover one algorithm of BFS type in the upcoming section.

VI. The general TREE-SEARCH algorithm is shown informally in Figure 2.Search algorithms all share this basic structure; they vary primarily according to how they choose which state to expand next—the so-called **search strategy**.

VII. The eagle-eyed reader will notice one peculiar thing about the search tree shown in Figure 1 it includes the path from Arad to Sibiu and back to Arad again! We say that In(Arad) is a **repeated state** in the search tree, generated in this case by a **loopy path**.

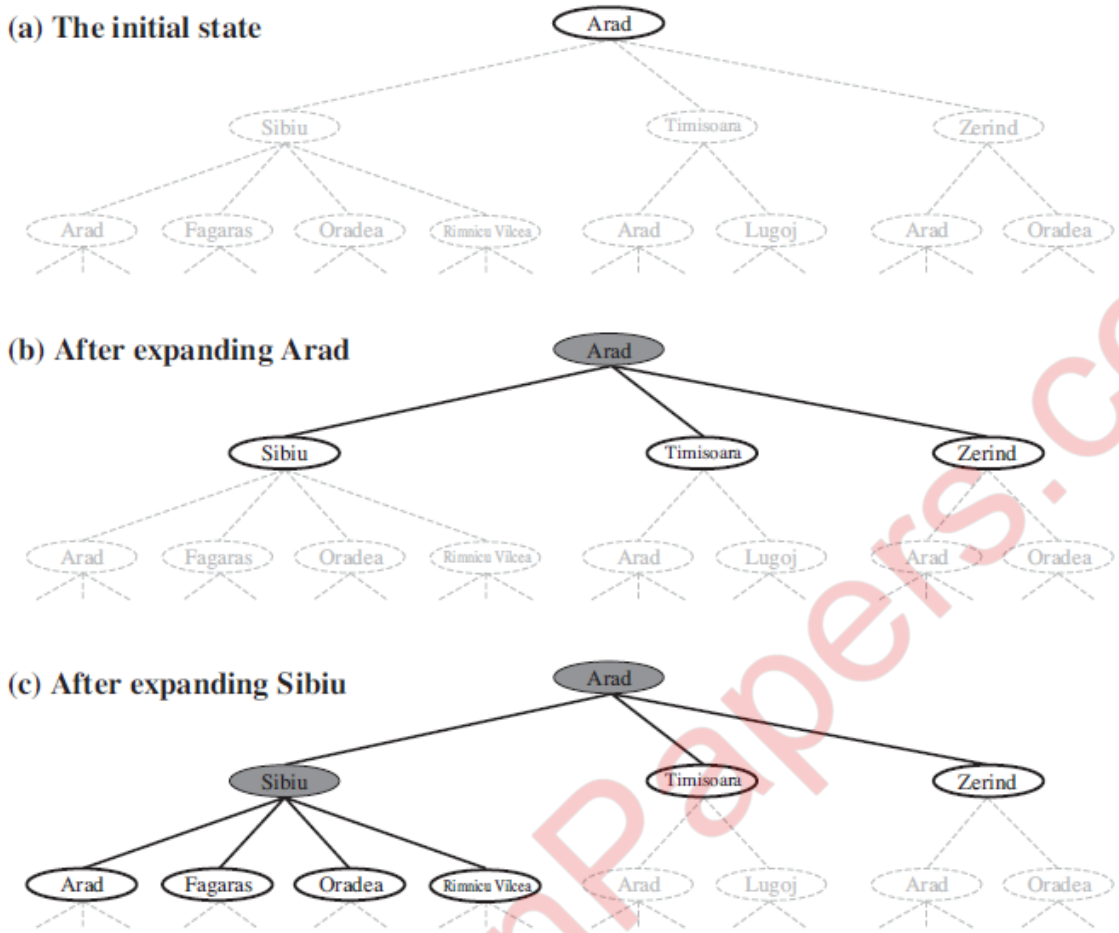


Figure 1 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

VIII. Loopy paths are a special case of the more general concept of **redundant paths**, which exist whenever there is more than one way to get from one state to another. Consider the paths Arad–Sibiu (140 km long) and Arad–Zerind–Oradea–Sibiu (297 km long). Obviously, the second path is redundant—it's just a worse way to get to the same state. If you are concerned about reaching the goal, there's never any reason to keep more than one path to any given state, because any goal state that is reachable by extending one path is also reachable by extending the other.

```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier
```

function GRAPH-SEARCH(problem) **returns** a solution, or failure
initialize the frontier using the initial state of problem
initialize the explored set to be empty
loop do
if the frontier is empty **then return** failure
choose a leaf node and remove it from the frontier
if the node contains a goal state **then return** the corresponding solution
add the node to the explored set
expand the chosen node, adding the resulting nodes to the frontier
only if not in the frontier or explored set

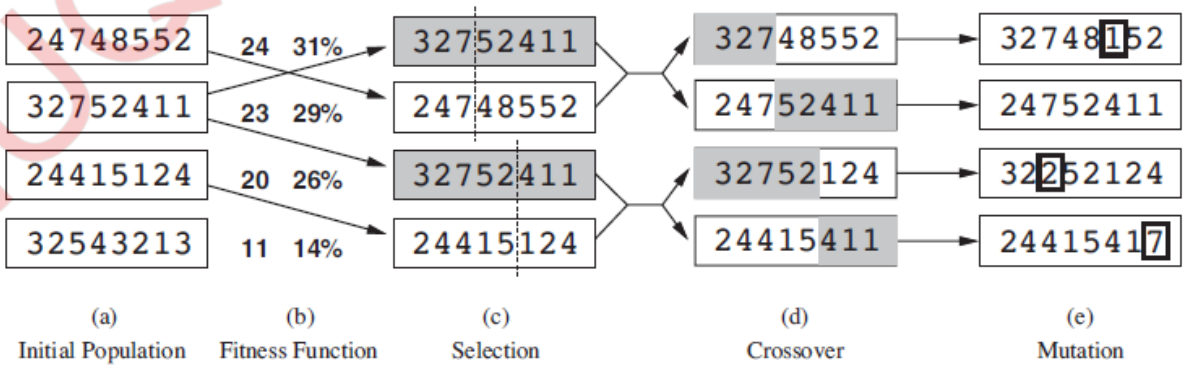
Figure 2 An informal description of the general tree-search and graph-search algorithms.

Q.2 d) Explain the mechanism of genetic algorithm. (5)

I. A **genetic algorithm** (or **GA**) is a variant of stochastic beam search in which successor states are generated by combining two parent states rather than by modifying a single state. The analogy to natural selection is the same as in stochastic beam search, except that now we are dealing with sexual rather than asexual reproduction.

II. Like beam searches, GAs begin with a set of k randomly generated states, called the population. Each state, or individual, is represented as a string over a finite alphabet—most commonly, a string of 0s and 1s.

III. For example, an 8-queens state must specify the positions of 8 queens, each in a column of 8 squares, and so requires 8× log2 8=24 bits. Alternatively, the state could be represented as 8 digits, each in the range from 1 to 8. (We demonstrate later that the two encodings behave differently.) Figure shows a population of four 8-digit strings representing 8-queens states.



The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs formating in (c). They produce offspring in (d), which are subject to mutation in (e).

IV. The following outline how the genetic algorithm works:

1. The algorithm begins by creating a random initial population.
2. The algorithm then creates a sequence of new populations. At each step, the algorithm uses the individuals in the current generation to create the next population. To create the new population, the algorithm performs the following steps:
    a. Scores each member of the current population by computing its fitness value. These values are called the raw fitness scores.
    b. Scales the raw fitness scores to convert them into a more usable range of values. These scaled values are called expectation values.
    c. Selects members, called parents, based on their expectation.
    d. Some of the individuals in the current population that have lower fitness are chosen as elite. These elite individuals are passed to the next population.
    e. Produces children from the parents. Children are produced either by making random changes to a single parent—mutation—or by combining the vector entries of a pair of parents—crossover.
    f. Replaces the current population with the children to form the next generation.
3. The algorithm stops when one of the stopping criteria is met.

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
    inputs: population, a set of individuals
            FITNESS-FN, a function that measures the fitness of an individual
    repeat
            new population ←empty set
            for i = 1 to SIZE(population) do
                x ←RANDOM-SELECTION(population, FITNESS-FN)
                y ←RANDOM-SELECTION(population, FITNESS-FN)
                child ←REPRODUCE(x , y)
                if (small random probability) then child ←MUTATE(child )
                add child to new population
            population ←new population
    until some individual is fit enough, or enough time has elapsed
    return the best individual in population, according to FITNESS-FN

function REPRODUCE(x , y) returns an individual
    inputs: x , y, parent individuals

    n←LENGTH(x ); c←random number from 1 to n
    return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

A genetic algorithm. The algorithm is the same as the one diagrammed in Figure, with one variation: in this more popular version, each mating of two parents produces only one offspring, not two.

V. Initial Population: - The algorithm begins by creating a random initial population,

VI. Creating the Next Generation:-

The genetic algorithm creates three types of children for the next generation:

- Eliteare the individuals in the current generation with the best fitness values. These individuals automatically survive to the next generation.
- Crossover are created by combining the vectors of a pair of parents.
- Mutation children are created by introducing random changes, or mutations, to a single parent.
    - **Crossover Children**

      The algorithm creates crossover children by combining pairs of parents in the current population. At each coordinate of the child vector, the default crossover function randomly selects an entry, or gene, at the same coordinate from one of the two parents and assigns it to the child. For problems with linear constraints, the default crossover function creates the child as a random weighted average of the parents.
    - **Mutation Children**

      The algorithm creates mutation children by randomly changing the genes of individual parents. By default, for unconstrained problems the algorithm adds a random vector from a Gaussian distribution to the parent. For bounded or linearly constrained problems, the child remains feasible.

VII. Plots of Later Generations

VIII. Stopping Conditions for the Algorithm

The genetic algorithm uses the following options to determine when to stop. See the default values for each option by running opts = optimoptions('ga').

- MaxGenerations —The algorithm stops when the number of generations reaches MaxGenerations.
- MaxTime —The algorithm stops after running for an amount of time in seconds equal to MaxTime.

Q.2 e) Explain how transition model is used for sensing in vacuum cleaner problem. (5)

Sensor-less search (condition)

- Transition model
    - Union of all states that $Result_p$(s) returns for all states, s, in your current belief state
        - $b' = Result(b,\ a)$ = $\{s' : s' = Result_p$(s, a) and s ϵ b$\}$
        - This is the prediction step, $Predict_p$(b, a)
- Goal-Test: If all physical states in belief state satisfy $Goal - Test_p$
- Path cost → Tricky in general. Consider what happens if actions in different physical states have different costs. For now assume cost of an action is the same in all states

1. **Prediction stage** is the same as for sensorless problems: given the action a in belief state b, the predicted belief state is ˆb = PREDICT(b, a)
2. **Observation prediction stage** determines the set of percepts o that could be observed in the predicted belief state:
   POSSIBLE−PERCEPTS(ˆ b) = {o : o = PERCEPT(s) and s ∈ ˆb}
3. **Update stage** determines, for each possible percept, the belief state that would result from the percept. The new belief state bo is just the set of states in ˆb that could have produced the percept: bo = UPDATE(ˆ b, o) = {s : o = PERCEPT(s) and s ∈ ˆb} the updated belief state bo can be no larger than the predicted belief state ˆb -the belief states for the different possible percepts will be disjoint, forming a partition of the original predicted belief state (for deterministic sensing).

Q.2 f) Give the illustration of 8 queen problem using hill climbing algorithm. (5)

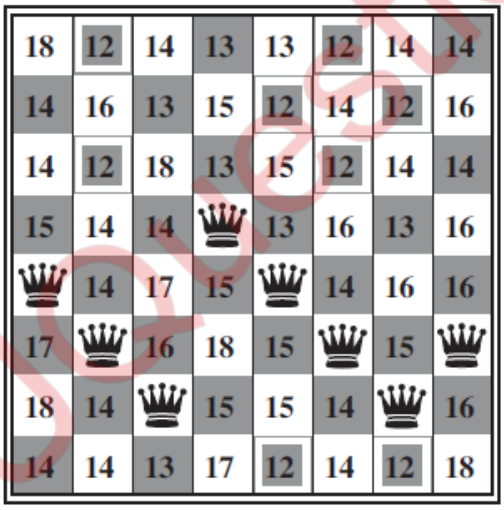
I. **The hill-climbing search algorithm** It is simply a loop that continually moves in the direction of increasing value—that is, uphill. It terminates when it reaches a "peak" where no neighbor has a higher value. The algorithm does not maintain a search tree, so the data structure for the current node need only record the state and the value of the objective function. Hill climbing does not look ahead beyond the immediate neighbors of the current state. This resembles trying to find the top of Mount Everest in a thick fog while suffering from amnesia.

II. Local search algorithms typically use a **complete-state formulation**, where each state has 8 queens on the board, one per column. The successors of a state are all possible states generated by moving a single queen to another square in the same column (so each state has 8×7=56 successors).

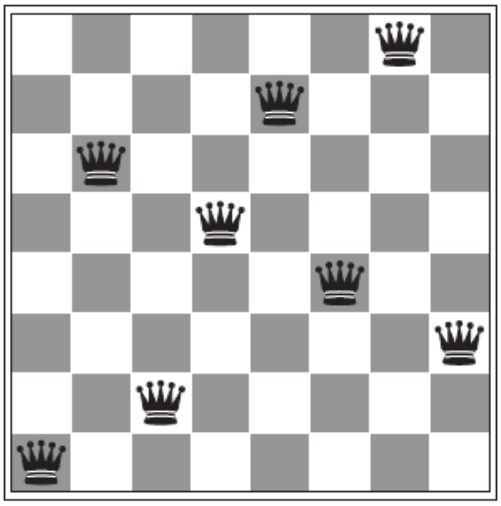
III. The heuristic cost function h is the number of pairs of queens that are attacking each other, either directly or indirectly. The global minimum of this function is zero, which occurs only at perfect solutions. Figure (a) shows a state with h=17. The figure also shows the values of all its successors, with the best successors having h=12.

IV. Hill-climbing algorithms typically choose randomly among the set of best successors if there is more than one. Hill climbing is sometimes called **greedy local search** because it grabs a good neighbour state without thinking ahead about where to go next. Although greed is considered one of the seven deadly sins, it turns out that greedy algorithms often perform quite well. Hill climbing often makes rapid progress toward a solution because it is usually quite easy to improve a bad state. For example, from the state in Figure (a), it takes just five steps to reach the state in Figure (b), which has h=1 and is very nearly a solution. Unfortunately, hill climbing often gets stuck for the following reasons:

- **Local maxima**: a local maximum is a peak that is higher than each of its neighbouring states but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go. More concretely, the state in Figure (b) is a local maximum (i.e., a local minimum for the cost h); every move of a single queen makes the situation worse.
- **Ridges**: Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
- **Plateaux**: a plateau is a flat area of the state-space landscape. It can be a flat local maximum, from which no uphill exit exists, or a **shoulder**, from which progress is possible. A hill-climbing search might get lost on the plateau.



(a)



(b)

(a) An 8-queens state with heuristic cost estimate h=17, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked.

(b) A local minimum in the 8-queens state space; the state has h=1 but every successor has a higher cost.

V. In each case, the algorithm reaches a point at which no progress is being made. Starting from a randomly generated 8-queens state, steepest-ascent hill climbing gets stuck 86% of the time, solving only 14% of problem instances. It works quickly, taking just 4 steps on average when it succeeds and 3 when it gets stuck—not bad for a state space with 8^8 ≈ 17 million states.

VI. If we always allow sideways moves when there are no uphill moves, an infinite loop will occur whenever the algorithm reaches a flat local maximum that is not a shoulder. One common solution is to put a limit on the number of consecutive sideways moves allowed. For example, we could allow up to, say, 100 consecutive sideways moves in the 8-queens problem. This raises the percentage of problem instances solved by hill climbing from 14% to 94%. Success comes at a cost: the algorithm averages roughly 21 steps for each successful instance and 64 for each failure.

VII. For 8-queens instances with no sideways moves allowed, p ≈ 0.14, so we need roughly 7 iterations to find a goal (6 failures and 1 success). The expected number of steps is the cost of one successful iteration plus (1−p)/p times the cost of failure, or roughly 22 steps in all. When we allow sideways moves, 1/0.94 ≈ 1.06 iterations are needed on average and (1×21) + (0.06/0.94)×64 ≈ 25 steps.

VIII. For 8-queens, then, random-restart hill climbing is very effective indeed. Even for three million queens, the approach can find solutions in under a minute.

IX. The success of hill climbing depends very much on the shape of the state-space landscape: if there are few local maxima and plateaux, random-restart hill climbing will find a good solution very quickly.

Q.3 a) Explain the working mechanism of min-max algorithm. (5)

I. Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn-based games such as Tic-Tac-Toe, Backgammon, Mancala, Chess, etc.

II. In Minimax the two players are called maximizer and minimizer.
The **maximizer** tries to get the highest score possible while the **minimizer** tries to do the opposite and get the lowest score possible.

III. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are **backed up** through the tree as the recursion unwinds.

IV. The minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is m and there are b legal moves at each point, then the time complexity of the minimax algorithm is O (bm).

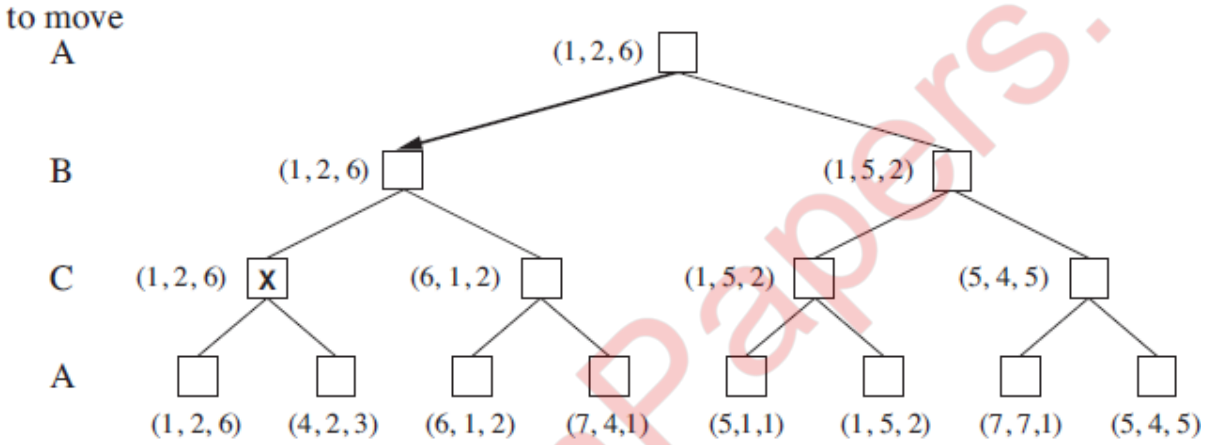
V. The space complexity is O (bm) for an algorithm that generates all actions at once, or O (m) for an algorithm that generates actions one at a time. For real games, of course, the time cost is totally impractical, but this algorithm serves as the basis for the mathematical analysis of games and for more practical algorithms.

Working of Min-Max Algorithm:

- The working of the minimax algorithm can be easily described using an example. Below we have taken an example of game-tree which is representing the two-player game.
- In this example, there are two players one is called Maximizer and other is called Minimizer.
- Maximizer will try to get the Maximum possible score, and Minimizer will try to get the minimum possible score.
- This algorithm applies DFS, so in this game-tree, we have to go all the way through the leaves to reach the terminal nodes.
- At the terminal node, the terminal values are given so we will compare those value and backtrack the tree until the initial state occurs. Following are the main steps involved in solving the two-player game tree:
- First, we need to replace the single value for each node with a vector of values. For example, in a three-player game with players A, B, and C, a vector (vA, vB, vC) is associated with each node. For terminal states, this vector gives the utility of the state from each player's viewpoint. (In two-player, zero-sum games, the two-element vector can be reduced to a single value because the values are always opposite.)
- The simplest way to implement this is to have the UTILITY function return a vector of utilities. Now we have to consider nonterminal states.
- Consider the node marked X in the game tree shown in Figure. In that state, player C chooses what to do. The two choices lead to terminal states with utility vectors (vA =1, vB =2, vC =6) and (vA =4, vB =2, vC =3). Since 6 is bigger than 3, C should choose the first move. This means that if state X is reached, subsequent play will lead to a terminal state with utilities (vA =1, vB =2, vC =6). Hence, the backed-up value of X is this vector. The backed-up value of a node n is always the utility vector of the successor state with the highest value for the player choosing at n.
- Anyone who plays multiplayer games, such as Diplomacy, quickly becomes aware that much more is going on than in two-player games. Multiplayer games usually involve **alliances**, whether formal or informal, among the players. Alliances are made and broken as the game proceeds. Strategies for each player in

a multiplayer game? It turns out that they can be. For example, suppose A and B are in weak positions and C is in a stronger position. Then it is often optimal for both A and B to attack C rather than each other, lest C destroy each of them individually. In this way, collaboration emerges from purely selfish behaviour.

- If the game is not zero-sum, then collaboration can also occur with just two players. Suppose, for example, that there is a terminal state with utilities _vA =1000, vB =1000_ and that 1000 is the highest possible utility for each player. Then the optimal strategy is for both players to do everything possible to reach this state—that is, the players will automatically cooperate to achieve a mutually desirable goal.



The first three plies of a game tree with three players (A, B, C). Each node is labelled with values from the viewpoint of each player. The best move is marked at the root.

Q.3 b) Explain in brief about resolution theorem. **(5)**

I. The process of forming an inferred clause or resolving from the parent clauses is called resolution.

II. This method demonstrates that the theorem being false causes an inconsistency with the axioms, hence the theorem must have been true all along. It uses only one rule of deduction, used to combine two **parent clauses** into a **resolved clause**.

III. We can express the full **resolution rule of inference** concisely using 'big $\vee$' notation: The 'big $\vee$' is just a more concise way of writing clauses, where underneath the V we specify a set of indices for the literals L. For example, if A = {1,2,7} then the first parent clause is $L_1 \vee L_2 \vee L_7$. (We can use a similar 'big $\wedge$' notation to express conjunctions.) The rule resolves literals P_j (a negative literal) and P_k (a positive literal). We just remove j and k from the set of indices to get the resolved clauses.

IV. We repeatedly resolve clauses until eventually two sentences resolve together to give the **empty clause**, which contains no literals.

- **Initial State**: A knowledge base (KB) consisting of negated theorem and axioms in CNF.
- **Operators**: The full resolution rule of inference picks two sentences from KB and adds a new sentence.
- **Goal Test**: Does KB contain False?

V. Illustrate the concept of a **resolution search space** with the simple example from Aristotle we've seen before. Apparently, all men are mortal and Socrates was a man. Given these words of wisdom, we want to prove that Socrates is mortal. We saw how this could be achieved using the Modus Ponens rule, and it is instructive to use Resolution to prove this as well.

- The initial KB (including the negated theorem) in CNF is:

  1) is_man(socrates)
  2) ¬is_man(X) ∨ is_mortal(X)
  3) ¬is_mortal(socrates)

- We can apply resolution to get TWO different solutions. The first alternative is that we combine (1) and (2) to get the state A:

  1) is_man(socrates)
  2) ¬is_man(X) ∨ is_mortal(X)
  3) ¬is_mortal(socrates)
  4) is_mortal(socrates)

- Then combine (3) and (4) to get the state B:

  1) is_man(socrates)
  2) ¬is_man(X) ∨ is_mortal(X)
  3) ¬is_mortal(socrates)
  4) is_mortal(socrates)
  5) False

- Alternatively, we could initially combine (2) and (3) to get the state C:

  1) is_man(socrates)
  2) ¬is_man(X) ∨ is_mortal(X)
  3) ¬is_mortal(socrates)
  4) ¬is_man(socrates)

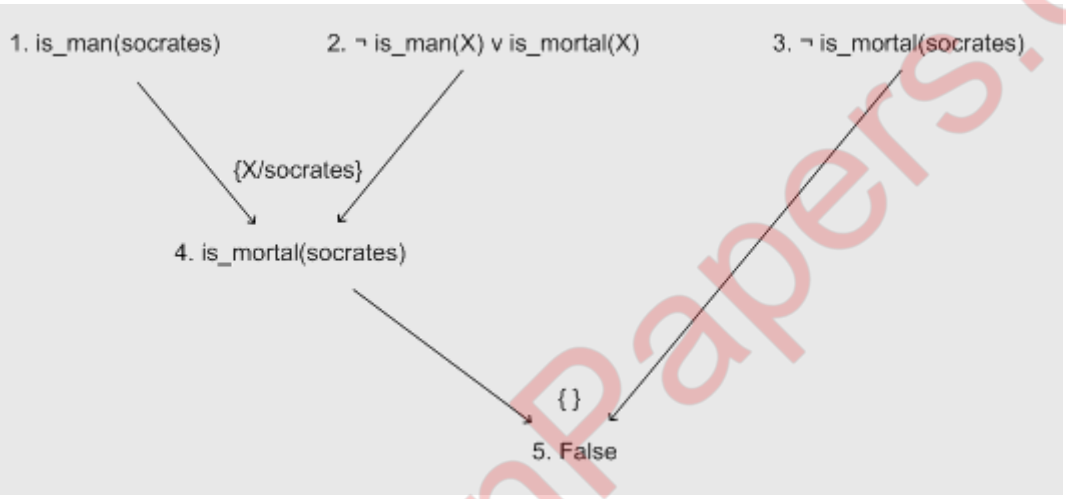
- We then resolve again to get state D:

  1) is_man(socrates)
  2) ¬is_man(X) ∨ is_mortal(X)
  3) ¬is_mortal(socrates)

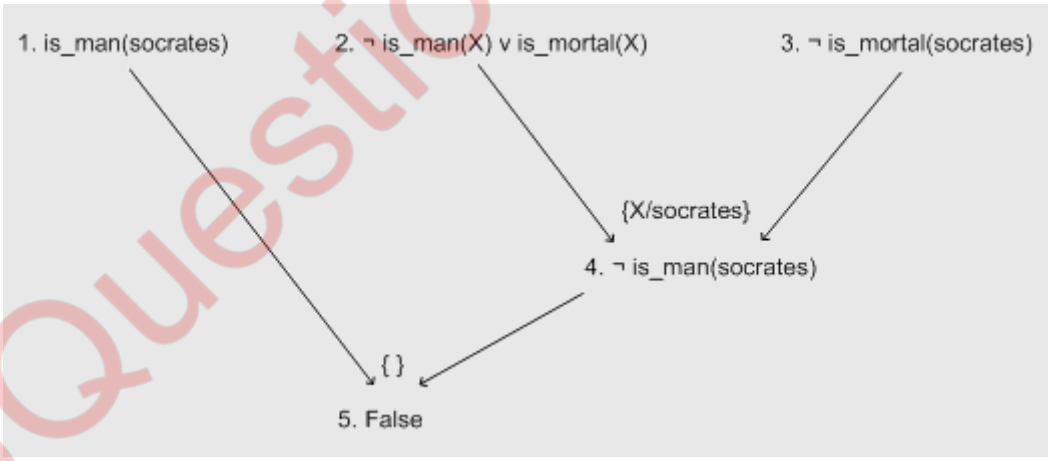
4) ¬is_man(socrates)
5) False

- So, we have a search space with two alternative paths to a solution: Initial --> A --> B and Initial --> C --> D.

VI. Instead, it is often more convenient to visualise the developing proof. On the top line we can write the clause of our initial KB, and draw lines from the two parent clauses to the new clause, indicating what substitution was required, if any. Repeating this process for each step we get a **proof tree**. Here's the finished proof tree for the path Initial --> A --> B in our example above:



An here's the proof tree for the alternative path Initial --> C --> D:



VII. Complex proofs require a bit effort to lay out, and it is ususally best not to write out all the initial clauses on the top line to begin with, but rather introduce them into the tree as they are required.

VIII. Resolution proof trees make it easier to recontruct a proof. Considering the latter tree, we can read the proof by working backwards from False. We could read the proof to Aristotle thus:

IX. "You said that all men were mortal. That means that for all things X, either X is not a man, or X is mortal [CNF step]. If we assume that Socrates is not mortal, then, given your previous statement, this means Socrates is not a man [first resolution step]. But you said that Socrates *is* a man, which means that our assumption was false [second resolution step], so Socrates must be mortal."

X. We see that, even in this simple case, it is difficult to translate the resolution proof into a human readable one. Due to the popularity of resolution theorem proving, and the difficulty with which humans read the output from the provers, there have been some projects to translate resolution proofs into a more human readable format. As an exercise, generate the proof you would give to Aristotle from the first proof tree.

XI. In the slides accompanying these notes is an example taken from Russell and Norvig about a cat called Tuna being killed by Curiosity. We will work through this example in the lecture.

Q.3 c) Write a note on Kriegspiel's Partially observable chess. (5)

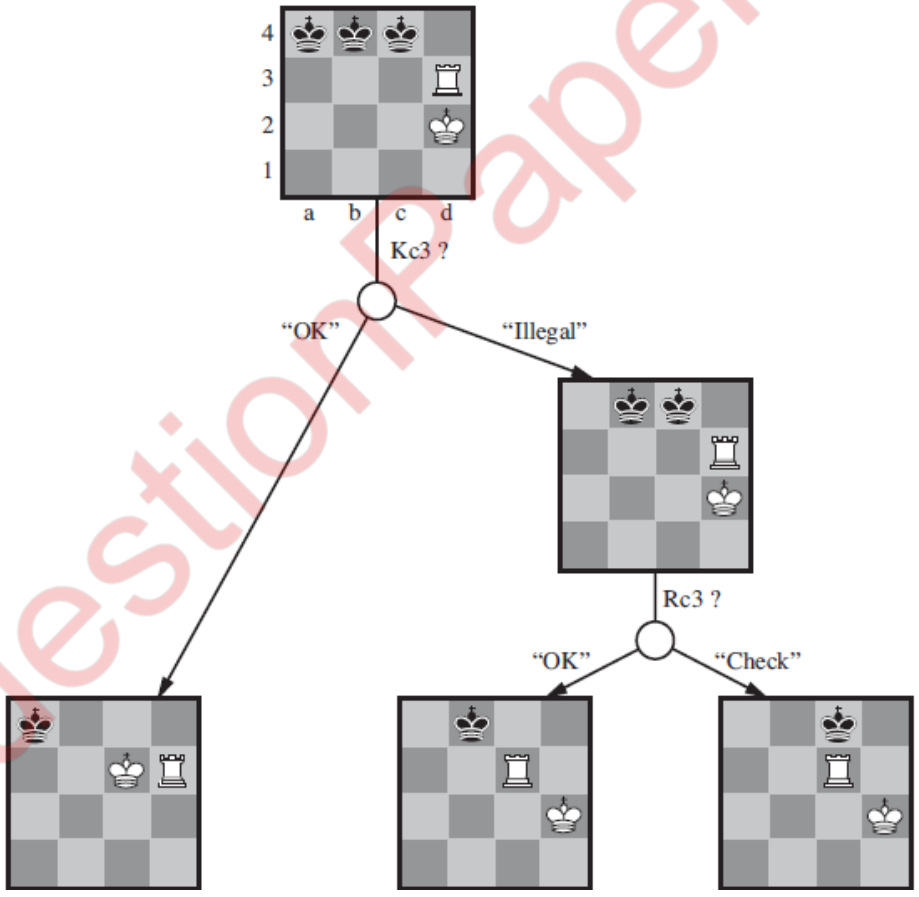
I. In deterministic partially observable games, uncertainty about the state of the board arises entirely from lack of access to the choices made by the opponent. This class includes children's games such as Battleships (where each player's ships are placed in locations hidden from the opponent but do not move) and Stratego (where piece locations are known but piece types are hidden). We will examine the game of **Kriegspiel**, a partially observable variant of chess in which pieces can move but are completely invisible to the opponent.

II. **The rules of Kriegspiel are as follows:** White and Black each see a board containing only their own pieces. A referee, who can see all the pieces, adjudicates the game and periodically makes announcements that are heard by both players. On his turn, White proposes to the referee any move that would be legal if there were no black pieces. If the move is in fact not legal (because of the black pieces), the referee announces "illegal." In this case, White may keep proposing moves until a legal one is found—and learns more about the location of Black's pieces in the process. Once a legal move is proposed, the referee announces one or more of the following: "Capture on square X" if there is a capture, and "Check by D" if the black king is in check, where D is the direction of the check, and can be one of "Knight," "Rank," "File," "Long diagonal," or "Short diagonal." (In case of discovered check, the referee may make two "Check" announcements.) If Black is checkmated or stalemated, the referee says so; otherwise, it is Black's turn to move.

III. Kriegspiel may seem terrifyingly impossible, but humans manage it quite well and computer programs are beginning to catch up. In Figure —the set of all logically possible board states given the complete history of percepts to date. Initially, White's belief state is a singleton because Black's pieces haven't moved yet. After White

makes a move and Black responds, White's belief state contains 20 positions because Black has 20 replies to any White move.

IV. Given a current belief state, White may ask, "Can I win the game?" For a partially observable game, the notion of a **strategy** is altered; instead of specifying a move to make for each possible move the opponent might make, we need a move for every possible percept sequence that might be received. For Kriegspiel, a winning strategy, or **guaranteed checkmate**, is one that, for each possible percept sequence, leads to an actual checkmate for every possible board state in the current belief state, regardless of how the opponent moves. With this definition, the opponent's belief state is irrelevant—the strategy has to work even if the opponent can see all the pieces. This greatly simplifies the computation. Figure shows part of a guaranteed checkmate for the KRK (king and rook against king) endgame. In this case, Black has just one piece (the king), so a belief state for White can be shown in a single board by marking each possible position of the Black king.



Part of a guaranteed checkmate in the KRK endgame, shown on a reduced board. In the initial belief state, Black's king is in one of three possible locations. By a combination of probing moves, the strategy narrows this down to one. Completion of the checkmate is left as an exercise.

V. The general AND-OR search algorithm can be applied to the belief-state space to find guaranteed checkmates, The incremental belief-state algorithm mentioned in that section often finds midgame checkmates up to depth 9—probably well beyond the abilities of human players.

VI. In addition to guaranteed checkmates, Kriegspiel admits an entirely new concept that makes no sense in fully observable games: probabilistic checkmate. Such checkmates are still required to work in every board state in the belief state; they are probabilistic with respect to randomization of the winning player's moves. To get the basic idea, consider the problem of finding a lone black king using just the white king. Simply by moving randomly, the white king will eventually bump into the black king even if the latter tries to avoid this fate, since Black cannot keep guessing the right evasive moves indefinitely. In the terminology of probability theory, detection occurs with probability 1. The KBNK endgame—king, bishop and knight against king—is won in this sense; White presents Black with an infinite random sequence of choices, for one of which Black will guess incorrectly and reveal his position, leading to checkmate. The KBBK endgame, on the other hand, is won with probability 1 – E.

VII. White can force a win only by leaving one of his bishops unprotected for one move. If Black happens to be in the right place and captures the bishop (a move that would lose if the bishops are protected), the game is drawn. White can choose to make the risky move at some randomly chosen point in the middle of a very long sequence, thus reducing _ to an arbitrarily small constant, but cannot reduce _ to zero.

VIII. It is quite rare that a guaranteed or probabilistic checkmate can be found within any reasonable depth, except in the endgame. Sometimes a checkmate strategy works for some of the board states in the current belief state but not others. Trying such a strategy may succeed, leading to an accidental checkmate—accidental in the sense that White could not know that it would be checkmate—if Black's pieces happen to be in the right places. (Most checkmates in games between humans are of this accidental nature.) This idea leads naturally to the question of how likely it is that a given strategy will win, which leads in turn to the question of how likely it is that each board state in the current belief state is the true board state.

IX. One's first inclination might be to propose that all board states in the current belief state are equally likely—but this can't be right. Consider, for example, White's belief state after Black's first move of the game. By definition (assuming that Black plays optimally), Black must have played an optimal move, so all board states resulting from suboptimal moves ought to be assigned zero probability. This argument is not quite right either, because each player's goal is not just to move pieces to the right squares but also to minimize the information that the opponent has about their location. Playing any predictable "optimal" strategy provides the opponent with information. Hence, optimal play in partially observable games requires a willingness to play somewhat randomly. (This is why restaurant hygiene inspectors do random inspection visits.) This means occasionally selecting moves that may seem "intrinsically" weak—but they gain strength from their very unpredictability, because the opponent is unlikely to have prepared any defense against them.

X. From these considerations, it seems that the probabilities associated with the board states in the current belief state can only be calculated given an optimal randomized strategy; in turn, computing that strategy seems to require knowing the probabilities of the various states the board might be in. This conundrum can be resolved by

adopting the game theoretic notion of an **equilibrium** solution. An equilibrium specifies an optimal randomized strategy for each player. Computing equilibria is prohibitively expensive, however, even for small games, and is out of the question for Kriegspiel. At present, the design of effective algorithms for general Kriegspiel play is an open research topic. Most systems perform bounded-depth lookahead in their own belief state space, ignoring the opponent's belief state. Evaluation functions resemble those for the observable game but include a component for the size of the belief state—smaller is better!

Q.3 d) Explain in brief about knowledge base agent. (5)

I. Knowledge is the basic element for a human brain to know and understand the things logically. When a person becomes knowledgeable about something, he is able to do that thing in a better way. In AI, the agents which copy such an element of human beings are known as knowledge-based agents.

II. The central component of a knowledge-based agent is its knowledge **base**, or KB. A knowledge base is a set of **sentences**. (Here "sentence" is used as a technical term. It is related but not identical to the sentences of English and other natural languages.) Each sentence is expressed in a language called a **knowledge representation language** and represents some assertion about the world. Sometimes we dignify a sentence with the name **axiom**, when the sentence is taken as given without being derived from other sentences.

III. There must be a way to add new sentences to the knowledge base and a way to query what is known. The standard names for these operations are TELL and ASK, respectively.

IV. Both operations may involve **inference**—that is, deriving new sentences from old. Inference must obey the requirement that when one ASKs a question of the knowledge base, the answer should follow from what has been told (or TELLed) to the knowledge base previously. Later in this chapter, we will be more precise about the crucial word "follow." For now, take it to mean that the inference process should not make things up as it goes along.

V. The agent maintains a knowledge base, KB, which may initially contain some **background knowledge**.

VI. Each time the agent program is called, it does three things. First, it TELLs the knowledge base what it perceives. Second, it ASKs the knowledge base what action it should perform. In the process of answering this query, extensive reasoning may be done about the current state of the world, about the outcomes of possible action sequences, and so on. Third, the agent program TELLs the knowledge base which action was chosen, and the agent executes the action.

VII. **Knowledge level**: - where we need specify only what the agent knows and what its goals are, in order to fix its behaviour. For example, an automated taxi might have the

goal of taking a passenger from San Francisco to Marin County and might know that the Golden Gate Bridge is the only link between the two locations. Then we can expect it to cross the Golden Gate Bridge because it knows that that will achieve its goal.

VIII. **Implementation level**: - Notice that this analysis is independent of how the taxi works at the **implementation level**. It doesn't matter whether its geographical knowledge is implemented as linked lists or pixel maps, or whether it reasons by manipulating strings of symbols stored in registers or by propagating noisy signals in a network of neurons.

IX. Example of knowledge-based agents is **wumpus world.**

X. The Wumpus world is a simple world example to illustrate the worth of a knowledge-based agent and to represent knowledge representation. It was inspired by a video game **Hunt the Wumpus** by Gregory Yob in 1973.

XI. The Wumpus world is a cave which has 4/4 rooms connected with passageways. So there are total 16 rooms which are connected with each other. We have a knowledge-based agent who will go forward in this world. The cave has a room with a beast which is called Wumpus, who eats anyone who enters the room. The Wumpus can be shot by the agent, but the agent has a single arrow. In the Wumpus world, there are some Pits rooms which are bottomless, and if agent falls in Pits, then he will be stuck there forever. The exciting thing with this cave is that in one room there is a possibility of finding a heap of gold. So the agent goal is to find the gold and climb out the cave without fallen into Pits or eaten by Wumpus. The agent will get a reward if he comes out with gold, and he will get a penalty if eaten by Wumpus or falls in the pit.

Q.3 e) Explain the syntax for propositional logic. (5)

- The syntax of propositional logic defines the allowable sentences. The atomic sentences consist of a single proposition symbol. Each such symbol stands for a proposition that can be true or false.
- We use symbols that start with an uppercase letter and may contain other letters or subscripts, for example: P, Q, R, W1, 3 and North. The names are arbitrary but are often chosen to have some mnemonic value—we use W1, 3 to stand for the proposition that the wumpus is in [1, 3]. (Remember that symbols such as W1, 3 are atomic, i.e., W, 1, and 3 are not meaningful parts of the symbol.) There are two proposition symbols with fixed meanings: True is the always-true proposition and False is the always-false proposition.
- Complex sentences are constructed from simpler sentences, using parentheses and logical connectives. There are five connectives in common use:

  1) ¬ (not):-

  A sentence such as ¬W1, 3 is called the negation of W1, 3. A literal is either an atomic sentence (a positive literal) or a negated atomic sentence (a negative literal).
  Example: - **¬A**

2) **∧ (and):-**
A sentence whose main connective is ∧, such as W1, 3 ∧ P3, 1, is called a conjunction; its parts are the conjuncts. (The ∧ looks like an "A" for "And.")
Example: - **A∧B**

3) **∨ (or):-**
A sentence using ∨, such as (W1, 3∧P3, 1) ∨W2, 2, is a **disjunction** of the **disjuncts** (W1, 3 ∧ P3, 1) and W2, 2. (Historically, the ∨ comes from the Latin "vel," which means "or." For most people, it is easier to remember ∨ as an upside-down ∧.)
Example: - **A∨B**

4) **⇒ (implies):-**
A sentence such as (W1, 3∧P3, 1) ⇒ ¬W2, 2 is called an implication (or conditional). Its premise or antecedent is (W1, 3 ∧P3, 1), and its conclusion or consequent is ¬W2, 2. Implications are also known as rules or if–then statements. The implication RULES symbol is sometimes written as ⊃ or →.
Example: - **A⇒B**

5) **⇔ (if and only if):-**
The sentence W1, 3 ⇔ ¬W2, 2 is a **biconditional**. In other way write this as ≡.
Example:- **A⇔B**

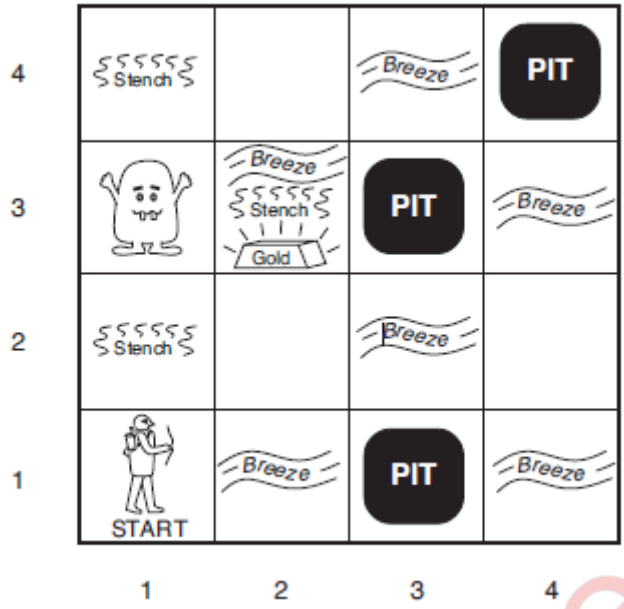
A	**B**	**A∧B**	**A∨B**	**¬A**	**A⇒B**	**A⇔B**
False	False	F	F	T	T	T
False	True	F	T	T	T	F
True	False	F	T	F	F	F
True	True	T	T	F	T	T

Q.3 f) Write a note on Wumpus world problem. **(5)**

I. The **wumpus world** is a cave consisting of rooms connected by passageways. Lurking somewhere in the cave is the terrible wumpus, a beast that eats anyone who enters its room. The wumpus can be shot by an agent, but the agent has only one arrow. Some rooms contain bottomless pits that will trap anyone who wanders into these rooms (except for the wumpus, which is too big to fall in). The only mitigating feature of this bleak environment is the possibility of finding a heap of gold. Although the wumpus world is rather tame by modern computer game standards, it illustrates some important points about intelligence.

II. A sample wumpus world is shown in Figure. The precise definition of the task environment is given, by the PEAS description:

- **Performance measure**: +1000 for climbing out of the cave with the gold, –1000 for falling into a pit or being eaten by the wumpus, –1 for each action taken and –10 for using up the arrow. The game ends either when the agent dies or when the agent climbs out of the cave.
- **Environment**: A 4×4 grid of rooms. The agent always starts in the square labelled [1, 1], facing to the right. The locations of the gold and the wumpus are chosen randomly, with a uniform distribution, from the squares other than the start square. In addition, each square other than the start can be a pit, with probability 0.2.
- **Actuators**: The agent can move Forward, TurnLeft by 90◦, or TurnRight by 90◦. The agent dies a miserable death if it enters a square containing a pit or a live wumpus. (It is safe, albeit smelly, to enter a square with a dead wumpus.) If an agent tries to move forward and bumps into a wall, then the agent does not move. The action Grab can be used to pick up the gold if it is in the same square as the agent. The action Shoot can be used to fire an arrow in a straight line in the direction the agent is facing. The arrow continues until it either hits (and hence kills) the wumpus or hits a wall. The agent has only one arrow, so only the first Shoot action has any effect. Finally, the action Climb can be used to climb out of the cave, but only from square [1, 1].
- **Sensors**: The agent has five sensors, each of which gives a single bit of information:
  – In the square containing the wumpus and in the directly (not diagonally) adjacent squares, the agent will perceive a Stench.
  – In the squares directly adjacent to a pit, the agent will perceive a Breeze.
  – In the square where the gold is, the agent will perceive a Glitter.
  – When an agent walks into a wall, it will perceive a Bump.
  – When the wumpus is killed, it emits a woeful Scream that can be perceived anywhere in the cave.



A typical wumpus world. The agent is in the bottom left corner, facing right.

III. The percepts will be given to the agent program in the form of a list of five symbols; for example, if there is a stench and a breeze, but no glitter, bump, or scream, the agent program will get [Stench, Breeze, None, None, None]. The wumpus environment along the various dimensions. Clearly, it is discrete, static, and single-agent. (The wumpus doesn't move, fortunately.) It is sequential, because rewards may come only after many actions are taken. It is partially observable, because some aspects of the state are not directly perceivable: the agent's location, the wumpus's state of health, and the availability of an arrow.

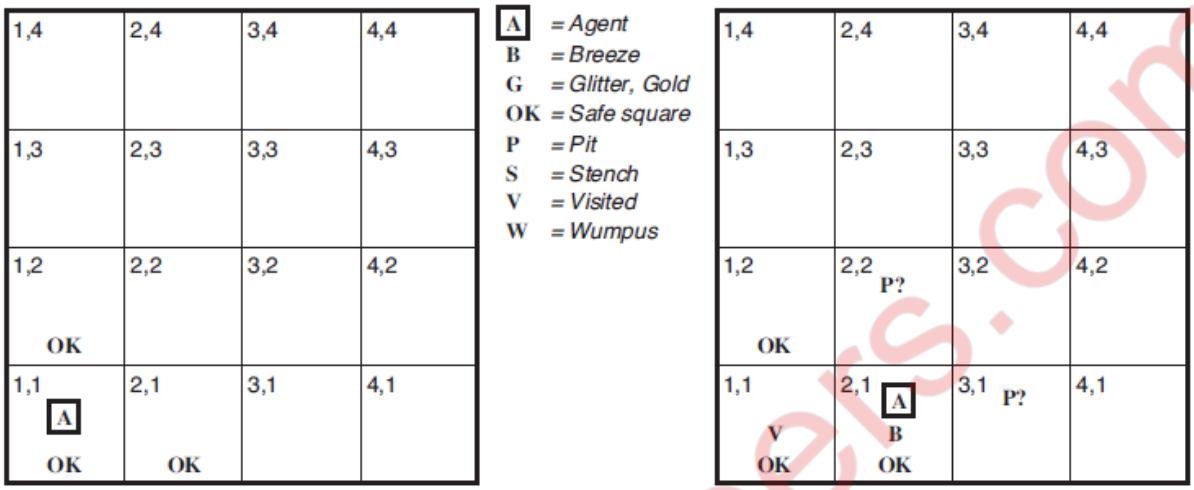
IV. As for the locations of the pits and the wumpus: we could treat them as unobserved parts of the state that happen to be immutable—in which case, the transition model for the environment is completely known; or we could say that the transition model itself is unknown because the agent doesn't know which Forward actions are fatal—in which case, discovering the locations of pits and wumpus completes the agent's knowledge of the transition model.

Exploring the problem of wumpus world:

I. We use an informal knowledge representation language consisting of writing down symbols in a grid (as in Figures 1 and 2).The agent's initial knowledge base contains the rules of the environment, as described previously; in particular, it knows that it is in [1, 1] and that [1, 1] is a safe square; we denote that with an "A" and "OK," respectively, in square [1, 1].

II. The first percept is [None, None, None, None, None], from which the agent can conclude that its neighboring squares, [1, 2] and [2, 1], are free of dangers—they are OK. Figure 1(a) shows the agent's state of knowledge at this point. A cautious agent will move only into a square that it knows to be OK. Let us suppose the agent decides to move forward to [2, 1]. The agent perceives a breeze (denoted by "B") in [2, 1], so there must be a pit in a neighboring square. The pit cannot be in [1, 1], by the rules of

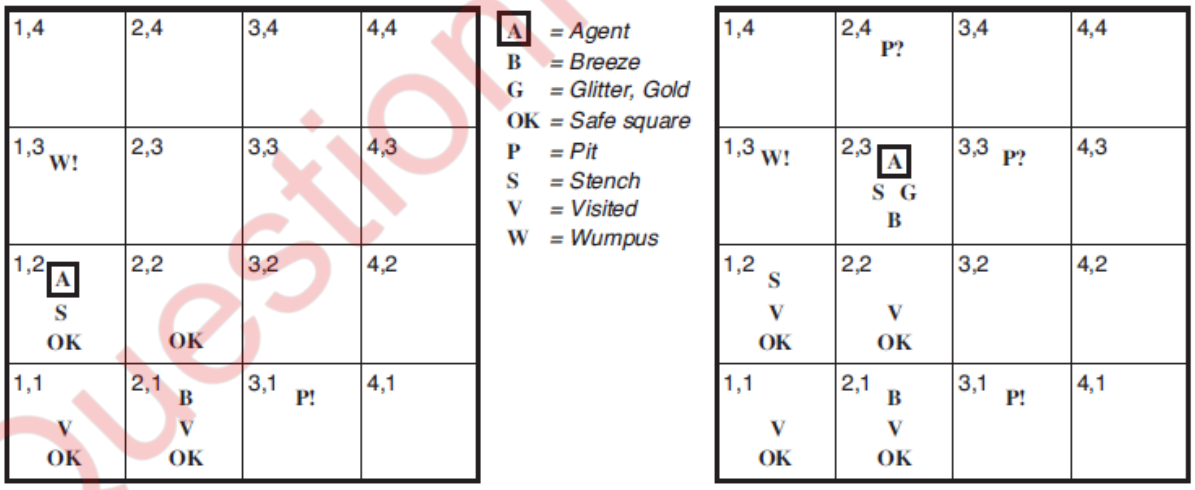
the game, so there must be a pit in [2, 2] or [3, 1] or both. The notation "P?" in Figure 1(b) indicates a possible pit in those squares. At this point, there is only one known square that is OK and that has not yet been visited. So the prudent agent will turn around, go back to [1, 1], and then proceed to [1, 2].



(a) (b)

1(a) **1(b)**

The first step taken by the agent in the wumpus world. (a) The initial situation, after percept [None, None, None, None, None]. (b) After one move, with percept [None, Breeze, None, None, None].



(a) (b)

2(a) **2(b)**

Two later stages in the progress of the agent. (a) After the third move, with percept [Stench, None, None, None, None]. (b) After the fifth move, with percept [Stench, Breeze, Glitter, None, None].

III. The agent perceives a stench in [1, 2], resulting in the state of knowledge shown in Figure 2(a). The stench in [1, 2] means that there must be a wumpus nearby. But the wumpus cannot be in [1, 1], by the rules of the game, and it cannot be in [2, 2] (or the agent would have detected a stench when it was in [2, 1]). Therefore, the agent can infer that the wumpus is in [1, 3]. The notation W! Indicates this inference. Moreover,

the lack of a breeze in [1, 2] implies that there is no pit in [2, 2]. Yet the agent has already inferred that there must be a pit in either [2, 2] or [3, 1], so this means it must be in [3, 1]. This is a fairly difficult inference, because it combines knowledge gained at different times in different places and relies on the lack of a percept to make one crucial step.

IV. The agent has now proved to itself that there is neither a pit nor a wumpus in [2, 2], so it is OK to move there. We do not show the agent's state of knowledge at [2, 2]; we just assume that the agent turns and moves to [2, 3], giving us Figure 2(b). In [2, 3], the agent detects a glitter, so it should grab the gold and then return home.

V. Note that in each case for which the agent draws a conclusion from the available information, that conclusion is guaranteed to be correct if the available information is correct.

VI. This is a fundamental property of logical reasoning. In the rest of this chapter, we describe how to build logical agents that can represent information and draw conclusions such as those described in the preceding paragraphs.

Q.4 a) What is first order logic? Discuss the different elements used in first order logic. **(5)**

I. First-order logic is another way of knowledge representation in artificial intelligence. It is an extension to propositional logic. FOL is sufficiently expressive to represent the natural language statements in a concise way.

II. First-order logic is also known as Predicate logic or First-order predicate logic. First-order logic is a powerful language that develops information about the objects in a more easy way and can also express the relationship between those objects.

III. First-order logic (like natural language) does not only assume that the world contains facts like propositional logic but also assumes the following things in the world:

- **Objects:** A, B, people, numbers, colors, wars, theories, squares, pits, wumpus,…
- **Relations:** It can be unary relation such as: red, round, is adjacent, or n-any relation such as: the sister of, brother of, has color, comes between
- **Function:** Father of, best friend, third inning of, end of,…

IV. As a natural language, first-order logic also has two main parts:

- **Syntax**
- **Semantics**

V. **Basic Elements of First-order logic:**

- Following are the basic elements of FOL syntax:

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

VI. Atomic sentences:

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

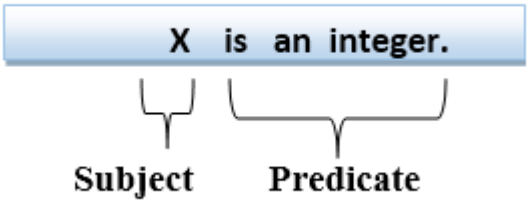
VII. Complex Sentences:

- Complex sentences are made by combining atomic sentences using connectives.

VIII. First-order logic statements can be divided into two parts:

- **Subject:** Subject is the main part of the statement.
- **Predicate:** A predicate can be defined as a relation, which binds two atoms together in a statement.

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



Q.4 b) Explain universal and existential quantifier with suitable example. (5)

- A logical quantifier that asserts all values of a given variable in a formula.
- First-order logic contains two standard quantifiers, called universal and existential.

1. Universal quantifier

- The symbol ∀ is called the **universal quantifier**.
- It expresses the fact that, in a particular universe of discourse, all objects have a particular property.
    - ∀x: means:
    - **For all objects** xx**, it is true that ...**
- ∀ is usually pronounced "For all . . .". (Remember that the upside-down A stands for "all.")
- That is:
- Thus, the sentence says, "For all x, if x is a king, then x is a person." The symbol x is called a variable. By convention, variables are lowercase letters. A variable is a term all by itself, and as such can also serve as the argument of a function—for example, LeftLeg(x). A term with no variables is called a ground term.
- The universal quantifier can be considered as a repeated conjunction:
- Suppose our universe of discourse consists of the objects X1, X2, X3…X1, X2, X3… and so on.

2. Existential quantifier

- The symbol ∃ is called the **existential quantifier**.
- It expresses the fact that, in a particular universe of discourse, there exists (at least one) object having a particular property.
  That is: ∃x means: **There exists at least one object** xx **such that ...**
- for example, that King John has a crown on his head, we write

  ∃ x Crown(x) ∧ OnHead(x, John) .
- ∃x is pronounced "There exists an x such that . . ." or "For some x . . ."

Q.4 c) Convert the following natural sentences into FOL form. (5)

i. Virat is cricketer.
Virat(cricketer)

ii. **All batsman are cricketers.**
For-all(x): batsman(x) -> cricketer(x)

iii. **Everybody speaks some language.**
For-all(x) Exist(y): Person(x) V language(y) -> speaks(x,y)

iv. **Every car has wheel.**
(forall (x) (if (Car x) (exists (y) wheel-of (x y)))

v. **Everybody loves somebody some time.**
(forall (x) (exists (y) -> loves-sometime(x y)))

Q.4 d) What is knowledge engineering? Write the steps for its execution. (5)

I. Knowledge engineering is a field of artificial intelligence (AI) that tries to emulate the judgment and behaviour of a human expert in a given field.

II. Knowledge engineering is the technology behind the creation of expert systems to assist with issues related to their programmed field of knowledge. Expert systems involve a large and expandable knowledge base integrated with a rules engine that specifies how to apply information in the knowledge base to each particular situation.

III. The systems may also incorporate machine learning so that they can learn from experience in the same way that humans do. Expert systems are used in various fields including healthcare, customer service, financial services, manufacturing and the law.

IV. Using algorithms to emulate the thought patterns of a subject matter expert, knowledge engineering tries to take on questions and issues as a human expert would. Looking at the structure of a task or decision, knowledge engineering studies how the conclusion is reached.

V. A library of problem-solving methods and a body of collateral knowledge are used to approach the issue or question. The amount of collateral knowledge can be very large. Depending on the task and the knowledge that is drawn on, the virtual expert may assist with troubleshooting, solving issues, assisting a human or acting as a virtual agent.

VI. Scientists originally attempted knowledge engineering by trying to emulate real experts. Using the virtual expert was supposed to get you the same answer as you would get from a human expert. This approach was called the transfer approach. However, the expertise that a specialist required to answer questions or respond to issues posed to it needed too much collateral knowledge: information that is not central to the given issue but still applied to make judgments.

VII. A surprising amount of collateral knowledge is required to enable analogous reasoning and nonlinear thought. Currently, a modelling approach is used where the same knowledge and process need not necessarily be used to reach the same conclusion for a given question or issue. Eventually, it is expected that knowledge engineering will produce a specialist that surpasses the abilities of its human counterparts.

Steps for knowledge engineering execution

Knowledge engineering projects vary widely in content, scope, and difficulty, but all such projects include the following steps:

I. **Identify the task**.
- The knowledge engineer must delineate the range of questions that the knowledge base will support and the kinds of facts that will be available for each specific problem instance.
- For example, does the wumpus knowledge base need to be able to choose actions or is it required to answer questions only about the contents of the environment? Will the sensor facts include the current location? The task will determine what knowledge must be represented in order to connect problem instances to answers.
- This step is analogous to the PEAS process for designing agents.

II. **Assemble the relevant knowledge**.
- The knowledge engineer might already be an expert in the domain, or might need to work with real experts to extract what they know—a process called knowledge acquisition.
- At this stage, the knowledge is not represented formally. The idea is to understand the scope of the knowledge base, as determined by the task, and to understand how the domain actually works.
- For the wumpus world, which is defined by an artificial set of rules, the relevant knowledge is easy to identify.
- For real domains, the issue of relevance can be quite difficult—for example, a system for simulating VLSI designs might or might not need to take into account stray capacitances and skin effects.

III. **Decide on a vocabulary of predicates, functions, and constants.**
- That is, translate the important domain-level concepts into logic-level names. This involves many questions of knowledge-engineering style.
- Like programming style, this can have a significant impact on the eventual success of the project. For example, should pits be represented by objects or by a unary predicate on squares? Should the agent's orientation be a function or a predicate? Should the wumpus's location depend on time? Once the

choices have been made, the result is a vocabulary that is known as the ontology of the domain.

- The word ontology means a particular theory of the nature of being or existence.
- The ontology determines what kinds of things exist, but does not determine their specific properties and interrelationships.

IV. Encode general knowledge about the domain.

- The knowledge engineer writes down the axioms for all the vocabulary terms. This pins down (to the extent possible) the meaning of the terms, enabling the expert to check the content.
- Often, this step reveals misconceptions or gaps in the vocabulary that must be fixed by returning to step 3 and iterating through the process.

V. Encode a description of the specific problem instance.

- If the ontology is well thought out, this step will be easy. It will involve writing simple atomic sentences about instances of concepts that are already part of the ontology.
- For a logical agent, problem instances are supplied by the sensors, whereas a "disembodied" knowledge base is supplied with additional sentences in the same way that traditional programs are supplied with input data.

VI. Pose queries to the inference procedure and get answers.

- This is where the reward is: we can let the inference procedure operate on the axioms and problem-specific facts to derive the facts we are interested in knowing.
- Thus, we avoid the need for writing an application-specific solution algorithm.

VII. Debug the knowledge base.

- Alas, the answers to queries will seldom be correct on the first try. More precisely, the answers will be correct for the knowledge base as written, assuming that the inference procedure is sound, but they will not be the ones that the user is expecting.
- For example, if an axiom is missing, some queries will not be answerable from the knowledge base. A considerable debugging process could ensue.
- Missing axioms or axioms that are too weak can be easily identified by noticing places where the chain of reasoning stops unexpectedly.

Q.4 e) Give comparison between forward chaining and backward chaining. (5)

Sr. No.	Forward chaining	Backward chaining
I.	Forward chaining starts from known facts and applies inference rule to extract more data unit it reaches to the goal.	Backward chaining starts from the goal and works backward through inference rules to find the required facts that support the goal.
II.	It is a bottom-up approach	It is a top-down approach
III.	Forward chaining is known as data-driven inference technique as we reach to the goal using the available data.	Backward chaining is known as goal-driven technique as we start from the goal and divide into sub-goal to extract the facts.
IV.	Forward chaining reasoning applies a breadth-first search strategy.	Backward chaining reasoning applies a depth-first search strategy.
V.	Forward chaining tests for all the available rules.	Backward chaining only tests for few required rules.
VI.	Forward chaining is suitable for the planning, monitoring, control, and interpretation application.	Backward chaining is suitable for diagnostic, prescription, and debugging application.
VII.	Forward chaining can generate an infinite number of possible conclusions.	Backward chaining generates a finite number of possible conclusions.
VIII.	It operates in the forward direction.	It operates in the backward direction.
IX.	Forward chaining is aimed for any conclusion.	Backward chaining is only aimed for the required data.

Q.4 f) Explain in brief about unification. (5)

I. Lifted inference rules require finding substitutions that make different logical expressions look identical. This process is called unification and is a key component of all first-order inference algorithms.

II. The UNIFY algorithm takes two sentences and returns a unifier for them if one exists:

- UNIFY(p, q)=θ where SUBST(θ, p)= SUBST(θ, q) .

III. Let us look at some examples of how UNIFY should behave. Suppose we have a query AskVars(Knows(John, x)): whom does John know? Answers to this query can be found by finding all sentences in the knowledge base that unify with Knows(John, x). Here are the results of unification with four different sentences that might be in the knowledge base:

- UNIFY(Knows(John, x), Knows(John, Jane)) = {x/Jane}
- UNIFY(Knows(John, x), Knows(y, Bill)) = {x/Bill, y/John}
- UNIFY(Knows(John, x), Knows(y,Mother (y))) = {y/John, x/Mother (John)}
- UNIFY(Knows(John, x), Knows(x, Elizabeth)) = fail .

IV. The last unification fails because x cannot take on the values John and Elizabeth at the same time. Now, remember that Knows(x, Elizabeth) means "Everyone knows Elizabeth," so we should be able to infer that John knows Elizabeth. The problem arises only because the two sentences happen to use the same variable name, x. The problem can be avoided by **standardizing apart** one of the two sentences being unified, which means renaming its variables to avoid name clashes. For example, we can rename x in Knows(x, Elizabeth) to x17 (a new variable name) without changing its meaning. Now the unification will work:

- UNIFY(Knows(John, x), Knows(x17, Elizabeth)) = {x/Elizabeth, x17/John} .

V. An algorithm for computing most general unifiers is shown in Figure. The process is simple: recursively explore the two expressions simultaneously "side by side," building up a unifier along the way, but failing if two corresponding points in the structures do not match.

VI. There is one expensive step: when matching a variable against a complex term, one must check whether the variable itself occurs inside the term; if it does, the match fails because no consistent unifier can be constructed. For example, S(x) can't unify with S(S(x)). This so called **occur check** makes the complexity of the entire algorithm quadratic in the size of the expressions being unified. Some systems, including all logic programming systems, simply omit the occur check and sometimes make unsound inferences as a result; other systems use more complex algorithms with linear-time complexity.

```
function UNIFY(x , y, θ) returns a substitution to make x and y identical
    inputs: x , a variable, constant, list, or compound expression
            y, a variable, constant, list, or compound expression
            θ, the substitution built up so far (optional, defaults to empty)

    if θ = failure then return failure
    else if x = y then return θ
    else if VARIABLE?(x ) then return UNIFY-VAR(x , y, θ)
    else if VARIABLE?(y) then return UNIFY-VAR(y, x , θ)
    else if COMPOUND?(x ) and COMPOUND?(y) then
            return UNIFY(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
    else if LIST?(x ) and LIST?(y) then
```

```
        return UNIFY(x .REST, y.REST, UNIFY(x .FIRST, y.FIRST, θ))
    else return failure

function UNIFY-VAR(var, x , θ) returns a substitution

    if {var/val} ∈ θ then return UNIFY(val , x , θ)
    else if {x/val} ∈ θ then return UNIFY(var, val , θ)
    else if OCCUR-CHECK?(var, x ) then return failure
    else return add {var/x } to θ
```

The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution è that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression such as F(A,B), the OP field picks out the function symbol F and the ARGS field picks out the argument list (A,B).

Q.5 a) What is planning? Explain STRIPS operators with suitable example. (5)

Planning

I. Artificial Intelligence is a critical technology in the future. Whether it is intelligent robots or self-driving cars or smart cities, they will all use different aspects of Artificial Intelligence!!! But to create any such AI project, Planning is very important. So much so that Planning is a critical part of Artificial Intelligence which deals with the actions and domains of a particular problem. Planning is considered as the reasoning side of acting.

II. For any planning system, we need the domain description, action specification, and goal description. A plan is assumed to be a sequence of actions and each action has its own set of preconditions to be satisfied before performing the action and also some effects which can be positive or negative.

III. The planning in Artificial Intelligence is about the decision making tasks performed by the robots or computer programs to achieve a specific goal.

IV. The execution of planning is about choosing a sequence of actions with a high likelihood to complete the specific task.

V. Planning is the fundamental management function, which involves deciding beforehand, what is to be done, when is it to be done, how it is to be done and who is going to do it. It is an intellectual process which lays down an organisation's objectives and develops various courses of action, by which the organisation can achieve those objectives. It chalks out exactly, how to attain a specific goal.

VI. Planning is nothing but thinking before the action takes place. It helps us to take a peep into the future and decide in advance the way to deal with the situations, which

we are going to encounter in future. It involves logical thinking and rational decision making.

Importance of Planning

I. It helps managers to improve future performance, by establishing objectives and selecting a course of action, for the benefit of the organisation.

II. It minimises risk and uncertainty, by looking ahead into the future.

III. It facilitates the coordination of activities. Thus, reduces overlapping among activities and eliminates unproductive work.

IV. It states in advance, what should be done in future, so it provides direction for action.

V. It uncovers and identifies future opportunities and threats.

VI. It sets out standards for controlling. It compares actual performance with the standard performance and efforts are made to correct the same.

STRIPS Operators

I. STRIPS Stands for STandford Research Institute Problem Solver

- Tidily arranged actions descriptions
- Restricted language (function-free literals)
- Efficient algorithms

II. **States** represented by:

Conjunction of ground (function-free) atoms

Example

At(Home), Have(Bread)

Closed world assumption

Atoms that are not present are assumed to be false

Example

State: At(Home), Have(Bread)

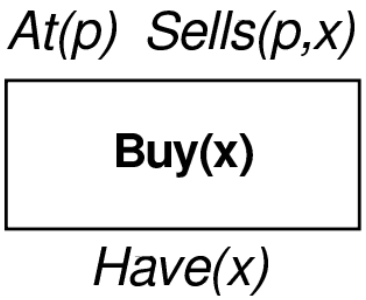
Implicitly: ¬Have(Milk),¬Have(Bananas),¬Have(Drill)

Operator description consists of:

Action name	Positive literal	Buy(Milk)
Precondition	Conjunction of positive literals	At(Shop)∧Sells(Shop,Milk)
Effect	Conjunction of literals	Have(Milk)

Operator schema

Operator containing variables



Operator applicability

Operator o applicable in state s if: there is substitution Subst of the free variables such that **Subst(precond(o)) ⊆ s**

Example

Buy(x) is applicable in state

At(Shop)∧Sells(Shop,Milk)∧Have(Bread)

with substitution

Subst = { p/Shop, x/Milk }

Resulting state

- Computed from old state and literals in Subst(effect)

1. Positive literals are added to the state
2. Negative literals are removed from the state
3. All other literals remain unchanged (avoids the frame problem)

Formally s' = (s ∪ {P | P a positive atom, P ∈ Subst(effect(o))})

\ {P | P a positive atom, ¬P ∈ Subst(effect(o))}

Example Application of

Drive(a,b) precond: At(a),Road(a,b) effect: At(b),¬At(a)

to state

At(Koblenz), Road(Koblenz,Landau)

results in

At(Landau), Road(Koblenz,Landau)

A complete set of STRIPS operators can be translated into a set of successor-state axioms

Q.5 b) Explain in brief about partially ordered plan. (5)

I. **Partial-order planning** is an approach to automated planning that maintains a partial ordering between actions and only commits ordering between actions when forced to i.e., ordering of actions is partial. Also this planning doesn't specify which action will come out first when two actions are processed.

II. Partially ordered plans are created by a search through the space of plans rather than through the state space.

III. By contrast, **total-order planning** maintains a total ordering between all actions at every stage of planning. Given a problem in which some sequence of actions is required in order to achieve a goal, a **partial-order plan** specifies all actions that need to be taken, but specifies an ordering between actions only where necessary.

IV. Consider the following situation: a person must travel from the start to the end of an obstacle course. This obstacle course is composed of a bridge, a see-saw and a swing-set. The bridge must be traversed before the see-saw and swing-set are reachable. Once reachable, the see-saw and swing-set can be traversed in any order, after which the end is reachable. In a partial-order plan, ordering between these obstacles is specified only when necessary. The bridge must be traversed first. Second, either the see-saw or swing-set can be traversed. Third, the remaining obstacle can be traversed. Then the end can be traversed. Partial-order planning relies upon the Principle of Least Commitment for its efficiency.

V. A **partial-order plan** or **partial plan** is a plan which specifies all actions that need to be taken, but only specifies the order between actions when necessary. It is the result of a partial-order planner. A partial-order plan consists of four components:

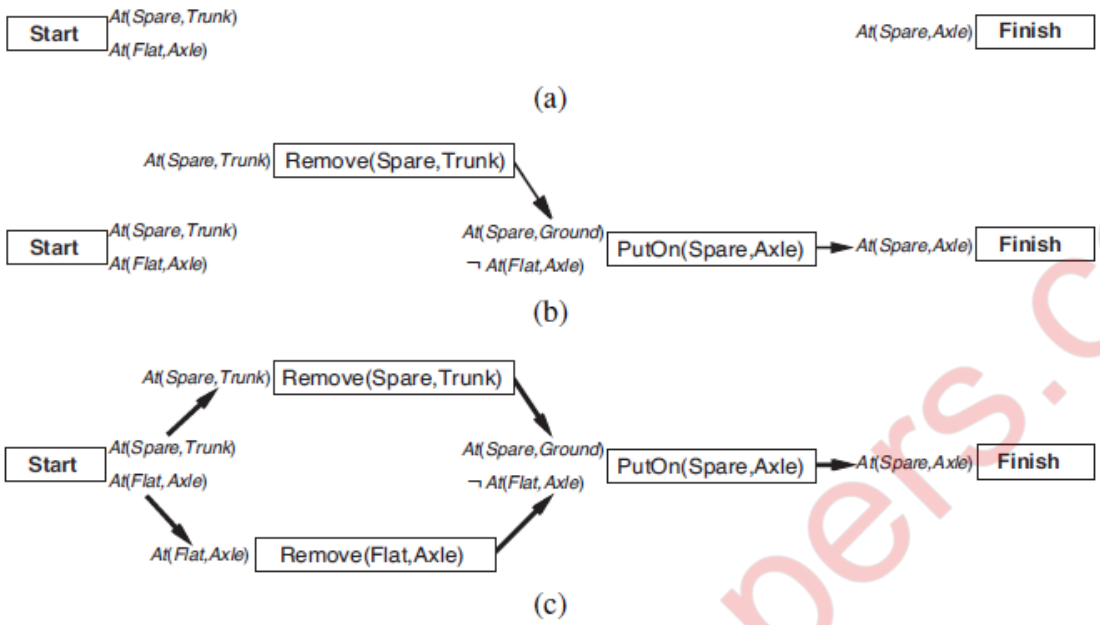
- **A set of actions** (also known as **operators**).
- **A partial order** for the actions. It specifies the conditions about the order of some actions.
- **A set of causal links**. It specifies which actions meet which preconditions of other actions. Alternatively, a set of **bindings** between the variables in actions.
- **A set of open preconditions**. It specifies which preconditions are not fulfilled by any action in the partial-order plan.

VI. In order to keep the possible orders of the actions as open as possible, the set of order conditions and causal links must be as small as possible.

VII. A plan is a solution if the set of open preconditions is empty.

VIII. A **linearization** of a partial order plan is a total order plan derived from the particular partial order plan; in other words, both order plans consist of the same actions, with

the order in the linearization being a linear extension of the partial order in the original partial order plan.



(a) The tire problem expressed as an empty plan. (b) An incomplete partially ordered plan for the tire problem. Boxes represent actions and arrows indicate that one action must occur before another. (c) A complete partially-ordered solution.

IX. The search keeps adding to the plan (backtracking if necessary) until all flaws are resolved, as in the bottom of Figure. At every step, we make the **least commitment** possible to fix the flaw. For example, in adding the action Remove(Spare, Trunk) we need to commit to having it occur before PutOn(Spare, Axle), but we make no other commitment that places it before or after other actions. If there were a variable in the action schema that could be left unbound, we would do so.

X. In the 1980s and 90s, partial-order planning was seen as the best way to handle planning problems with independent subproblems—after all, it was the only approach that explicitly represents independent branches of a plan. On the other hand, it has the **disadvantage** of not having an explicit representation of states in the state-transition model. That makes some computations cumbersome.

XI. Partial-order planners are not competitive on fully automated classical planning problems. However, partial-order planning remains an important part of the field. For some specific tasks, such as operations scheduling, partial-order planning with domain specific heuristics is the technology of choice.

XII. Partial-order planning is also often used in domains where it is important for humans to understand the plans. Operational plans for spacecraft and Mars rovers are generated by partial-order planners and are then checked by human operators before being uploaded to the vehicles for execution. The plan refinement approach makes it easier for the humans to understand what the planning algorithms are doing and verify that they are correct.

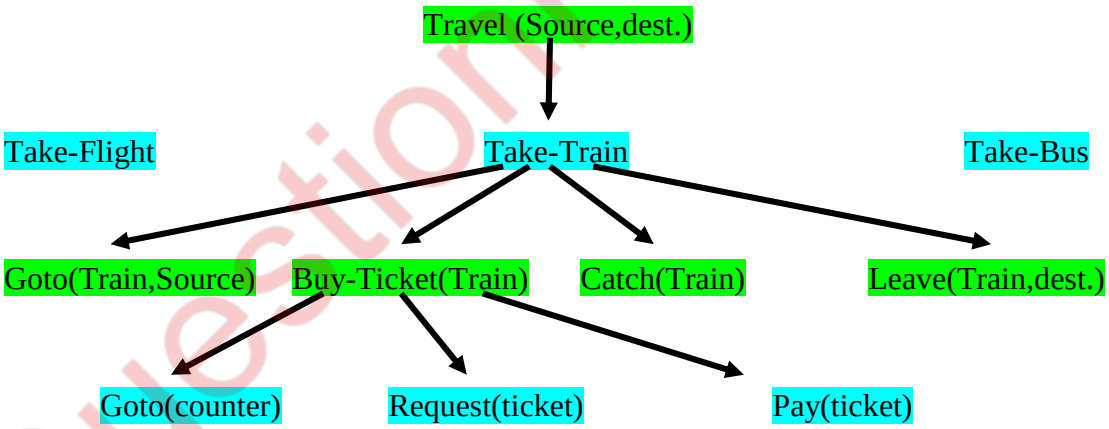
Q.5 c) Explain in brief about hierarchical planning. **(5)**

I. Ever since the conception of Artificial Intelligence, hierarchical problem solving has been used as a method to reduce the computational cost of planning.

II. The idea of hierarchical problem-solving, a well-accepted one, is to distinguish between goals and actions of different degrees of importance, and solve the most important problems first. Its main advantage derives from the fact that by emphasizing certain activities while temporarily ignoring others, it is possible to obtain a much smaller search space in which to find a plan.

III. As an example, suppose that in the household domain we would like to paint the ceiling white. Initially the number of conditions to consider may be overwhelming, ranging from the availability of various supplies, the suppliers for equipment and tools, to the position of the agent, the ladder, and the state of the ceiling. However, we could obtain a more manageable search space by first concentrating on whether we have the paint, the ladder, and a brush. Once a plan is found we then consider how to refine this plan by considering how to get to the rooms where each item is located. The process repeats until a full-blown plan is finally found.

IV. Figure shows, how to create a hierarchical plan to travel from some source to a destination.

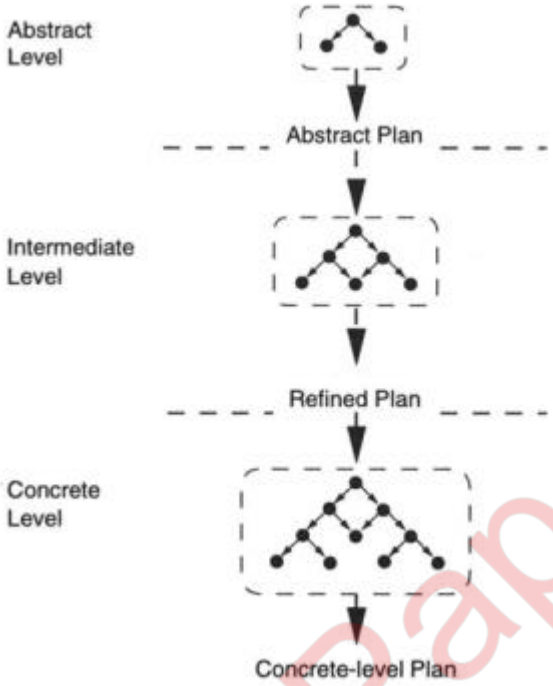


Hierarchical planning example

V. **A Hierarchical Planner**

- The intuition behind the operation of a hierarchical planner is shown in Figure In this figure there are three levels of abstraction, an abstract level, an intermediate level and a concrete level.
- Each dashed box represents a problem-solver at a given level.
- A planning problem is first abstracted and solved at the most abstract level. The solution obtained at this level, an abstract plan, is taken as the input to a problem-solver at the next level.

- The process ends when a concrete-level solution is found. In general, the abstraction levels could range from a single level to multiple levels. The former is identical to problem-solving without any abstraction.



Illustrating hierarchical planning

VI. Planner:-

- First identify a hierarchy of major conditions.
- Construct a plan in levels (Major steps then minor steps), so we postpone the details to next level.
- Patch major levels as detail actions become visible.
- Finally demonstrate.

VII. Example:-

- Actions required for "Travelling to Goa":
  - Opening makemytrip.com (1)
  - Finding flight (2)
  - Buy Ticket (3)
  - Get taxi(2)
  - Reach airport(3)
  - Pay-driver(1)
  - Check in(1)
  - Boarding plane(2)

- Reach Goa(3)

- 1st level Plan :
  - Buy Ticket (3), Reach airport(3), Reach Goa(3)
- 2nd level Plan :
  - Finding flight (2), Buy Ticket (3), Get taxi(2), Reach airport(3), Boarding plane(2), Reach Goa(3)
- 3rd level Plan (final) :
  - Opening makemytrip.com (1), Finding flight (2), Buy Ticket (3), Get taxi(2), Reach airport(3), Pay-driver(1), Check in(1), Boarding plane(2), Reach Goa(3)

Q.5 d) Write a note on mutex relation. (5)

I. A mutex relation holds between two actions at a given level if any of the following three conditions holds:
- **Inconsistent effects:** one action negates an effect of the other. For example, Eat (Cake) and the persistence of Have(Cake) have inconsistent effects because they disagree on the effect Have(Cake).
- **Interference:** one of the effects of one action is the negation of a precondition of the other. For example Eat (Cake) interferes with the persistence of Have(Cake) by negating its precondition.
- **Competing needs:** one of the preconditions of one action is mutually exclusive with a precondition of the other. For example, Bake(Cake) and Eat (Cake) are mutex because they compete on the value of the Have(Cake) precondition.

II. A mutex relation holds between two literals at the same level if one is the negation of the other or if each possible pair of actions that could achieve the two literals is mutually exclusive. This condition is called inconsistent support.

III. For example, Have(Cake) and Eaten(Cake) are mutex in S1 because the only way of achieving Have(Cake), the persistence action, is mutex with the only way of achieving Eaten(Cake), namely Eat (Cake).

IV. In S2 the two literals are not mutex, because there are new ways of achieving them, such as Bake(Cake) and the persistence of Eaten(Cake), that are not mutex.

V. A planning graph is polynomial in the size of the planning problem. For a planning problem with l literals and a actions, each Si has no more than l nodes and l^2 mutex links, and each Ai has no more than a + l nodes (including the no-ops), (a + l)^2 mutex links, and 2(al + l) precondition and effect links. Thus, an entire graph with n levels has a size of O(n(a + l)^2). The time to build the graph has the same complexity.

Q.5 e) What is semantic network? Show the semantic representation with suitable example. **(5)**

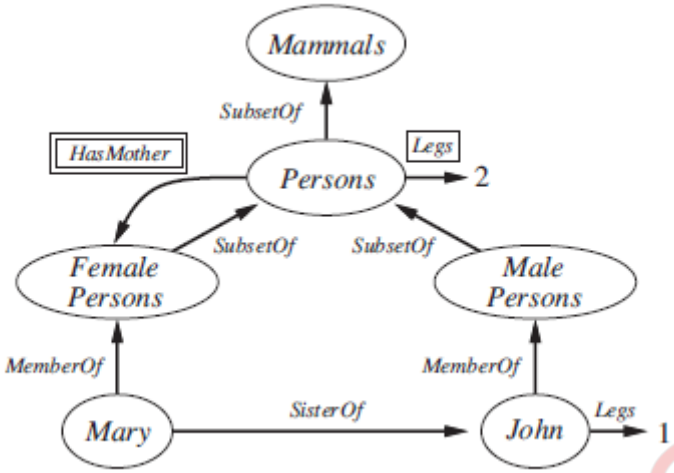
I. Semantic networks are an alternative to predicate logic as a form of knowledge representation. The idea is that we can store our knowledge in the form of a graph, with nodes representing objects in the world, and arcs representing relationships between those objects.

II. A **semantic network**, or **frame network** is a knowledge base that represents semantic relations between concepts in a network. It is a directed or undirected graph consisting of vertices, which represent concepts, and edges, which represent semantic relations between concepts, mapping or connecting semantic fields. A semantic network may be instantiated as, for example, a graph database or a concept map.

III. Typical standardized semantic networks are expressed as semantic triples. Semantic networks are used in natural language processing applications such as semantic parsing and word-sense disambiguation.

IV. The structural idea is that knowledge can be stored in the form of graphs, with nodes representing objects in the world, and arcs representing relationships between those objects.

- Semantic nets consist of nodes, links and link labels. In these networks diagram, nodes appear in form of circles or ellipses or even rectangles which represents objects such as physical objects, concepts or situations.
- Links appear as arrows to express the relationships between objects, and link labels specify relations.
- Relationships provide the basic needed structure for organizing the knowledge, so therefore objects and relations involved are also not needed to be concrete.
- Semantic nets are also referred to as associative nets as the nodes are associated with other nodes



A semantic network with four objects (John, Mary, 1, and 2) and four categories. Relations are denoted by labeled links.

V. For example, Figure has a MemberOf link between Mary and FemalePersons , corresponding to the logical assertion Mary ∈FemalePersons ; similarly, the SisterOf link between Mary and John corresponds to the assertion SisterOf (Mary, John). We can connect categories using SubsetOf links, and so on. It is such fun drawing bubbles and arrows that one can get carried away.

VI. For example, we know that persons have female persons as mothers, so can we draw a HasMother link from Persons to FemalePersons? The answer is no, because HasMother is a relation between a person and his or her mother, and categories do not have mother For this reason, we have used a special notation—the double-boxed link—in Figure This link asserts that

$$\forall x\ x \in \text{Persons} \Rightarrow [\forall y\ \text{HasMother}(x, y) \Rightarrow y \in \text{FemalePersons}]\ .$$

We might also want to assert that persons have two legs—that is,

$$\forall x\ x \in \text{Persons} \Rightarrow \text{Legs}(x, 2)$$

VII. Semantic Networks Are Majorly Used For

- Representing data
- Revealing structure (relations, proximity, relative importance)
- Supporting conceptual edition
- Supporting navigation

VIII. Advantages of Using Semantic Networks

- The semantic network is more natural than the logical representation;
- The semantic network permits using of effective inference algorithm (graphical algorithm)
- They are simple and can be easily implemented and understood.

- The semantic network can be used as a typical connection application among various fields of knowledge, for instance, among computer science and anthropology.
- The semantic network permits a simple approach to investigate the problem space.

IX. Disadvantages of Using Semantic Networks

- There is no standard definition for link names
- Semantic Nets are not intelligent, dependent on the creator

---

Q.5 f) Write a note on Event calculus. (5)

I. Situation calculus is limited in its applicability: it was designed to describe a world in which actions are discrete, instantaneous, and happen one at a time. Consider a continuous action, such as filling a bathtub. Situation calculus can say that the tub is empty before the action and full when the action is done, but it can't talk about what happens during the action. It also can't describe two actions happening at the same time—such as brushing one's teeth while waiting for the tub to fill. To handle such cases we introduce an alternative formalism known as **event calculus**, which is based on points of time rather than on situations.

II. Event calculus reifies fluents and events. The fluent At(Shankar , Berkeley) is an object that refers to the fact of Shankar being in Berkeley, but does not by itself say anything about whether it is true. To assert that a fluent is actually true at some point in time we use the predicate T, as in T(At(Shankar , Berkeley), t).

III. Events are described as instances of event categories. The event E1 of Shankar flying from San Francisco to Washington, D.C. is described as
E1 ∈ Flyings ∧ Flyer (E1, Shankar) ∧ Origin(E1, SF) ∧ Destination(E1,DC) .

IV. If this is too verbose, we can define an alternative three-argument version of the category of flying events and say
E1 ∈ Flyings(Shankar , SF,DC) .

V. We then use Happens(E1, i) to say that the event E1 took place over the time interval i, and we say the same thing in functional form with Extent(E1)=i. We represent time intervals by a (start, end) pair of times; that is, i = (t1, t2) is the time interval that starts at t1 and ends at t2.

VI. The complete set of predicates for one version of the event calculus is

• **T(f, t)**	Fluent f is true at time t
• **Happens(e, i)**	Event e happens over the time interval i
• **Initiates(e, f, t)**	Event e causes fluent f to start to hold at time t
• **Terminates(e, f, t)**	Event e causes fluent f to cease to hold at time t
• **Clipped(f, i)**	Fluent f ceases to be true at some point during time interval i
• **Restored (f, i)**	Fluent f becomes true sometime during time interval i

VII. We can extend event calculus to make it possible to represent simultaneous events (such as two people being necessary to ride a seesaw), exogenous events (such as the wind blowing and changing the location of an object), continuous events (such as the level of water in the bathtub continuously rising) and other complications.