Paper/ Subject Code: 53704 / Artificial Intelligence

(APR 2019) **Q.P. code: 57834**

Q.1 a) What is the purpose of turing test? **(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 b) What is Artificial Intelligence? Explain with example. **(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.

Examples of AI:-

1. **Alexa**

- Alexa's rise to become the smart home's hub, has been somewhat meteoric. When Amazon first introduced Alexa, it took much of the world by storm.
- However, it's usefulness and its uncanny ability to decipher speech from anywhere in the room has made it a revolutionary product that can help us scour the web for information, shop, schedule appointments, set alarms and a million other things, but also help power our smart homes and be a conduit for those that might have limited mobility.

2. **Amazon.com**

- Amazon's transactional A.I. is something that's been in existence for quite some time, allowing it to make astronomical amounts of money online.

- With its algorithms refined more and more with each passing year, the company has gotten acutely smart at predicting just what we're interested in purchasing based on our online behaviour.

3. Face Detection and Recognition

- Using virtual filters on our face when taking pictures and using face ID for unlocking our phones are two applications of AI that are now part of our daily lives.
- The former incorporates face detection meaning any human face is identified. The latter uses face recognition through which a specific face is recognised.

4. Chatbots

- As a customer, getting queries answered can be time-consuming. An artificially intelligent solution to this is the use of algorithms to train machines to cater to customers via chatbots.

- This enables machines to answer FAQs, and take and track orders.

5. Social Media

- The advent of social media provided a new narrative to the world with excessive freedom of speech.

- Various social media applications are using the support of AI to control these problems and provide users with other entertaining features.

- AI algorithms can spot and swiftly take down posts containing hate speech a lot faster than humans could. This is made possible through their ability to identify hate keywords, phrases, and symbols in different languages.

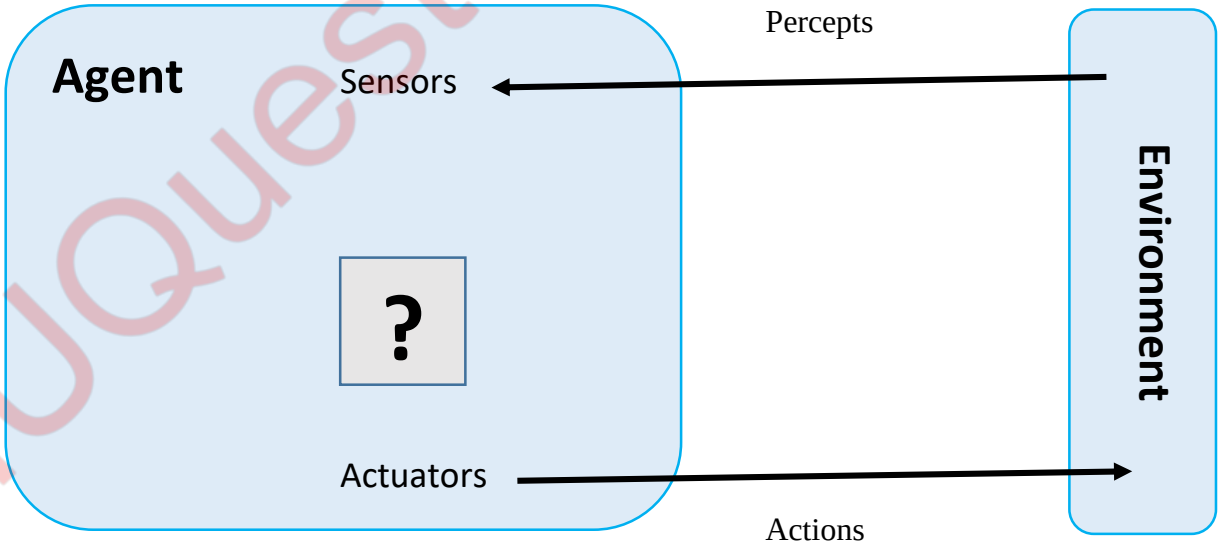
6. E-Payments

- Artificial intelligence has made it possible to deposit cheques from the comfort of your home. AI is proficient in deciphering handwriting, making online cheque processing practicable.

- The way fraud can be detected by observing users' credit card spending patterns is also an example of artificial intelligence.

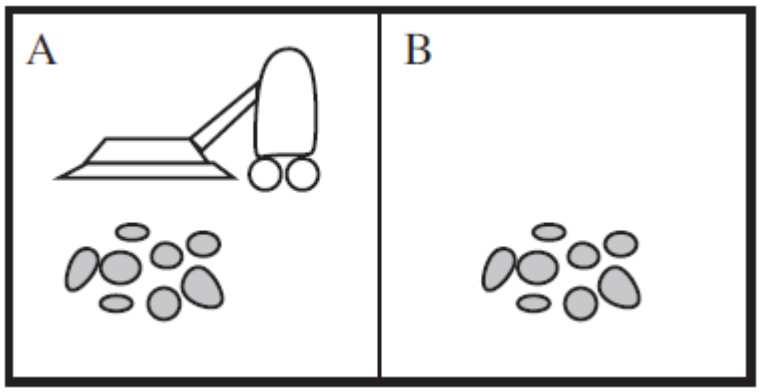
Q.1 c) Explain the concept of agent and environment. (5)

- 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) Give the PEAS description for taxi's task environment. (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 for taxi's task environment

- **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 e) Explain the rational agent approach of AI. (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 f) Explain the working of simple reflex agent. (5)

I. The simplest kind of agent is the **simple reflex agent**. These agents select actions on the basis of the current percept, ignoring the rest of the percept history.

II. A **simple reflex agent** is the most basic of the intelligent agents out there. It performs actions based on a current situation. When something happens in the environment of a simple reflex agent, the agent quickly scans its knowledge base for how to respond to the situation at-hand based on pre-determined rules.

III. It would be like a home thermostat recognizing that if the temperature increases to 75 degrees in the house, the thermostat is prompted to kick on. It doesn't need to know what happened with the temperature yesterday or what might happen tomorrow. Instead, it operates based on the idea that if _____ happens, _____ is the response.

IV. Simple reflex agents are just that: simple. They cannot compute complex equations or solve complicated problems. They work only in environments that are fully-observable in the current percept, ignoring any percept history. If you have a smart light bulb, for example, set to turn on at 6 p.m. every night, the light bulb will not recognize how the days are longer in summer and the lamp is not needed until much later. It will continue to turn the lamp on at 6 p.m. because that is the rule it follows. Simple reflex agents are built on the condition-action rule.

V. Simple reflex behaviors occur even in more complex environments. Imagine yourself as the driver of the automated taxi. If the car in front brakes and its brake lights come on, then you should notice this and initiate braking. In other words, some processing is done on the visual input to establish the condition we call "The car in front is braking." Then, this triggers some established connection in the agent program to the action "initiate braking." We call such a connection a **condition–action rule**, written as

if car-in-front-is-braking **then** initiate-braking.

VI. For example, the vacuum agent is a simple reflex agent, because its decision is based only on the current location and on whether that location contains dirt.

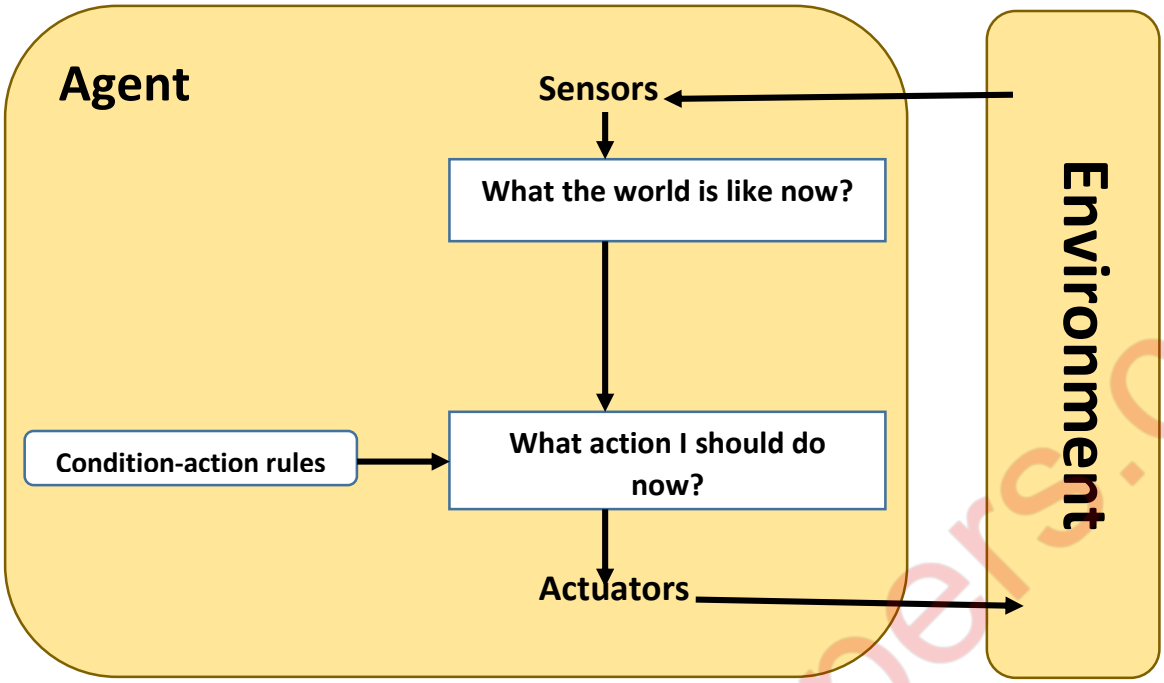
An agent program for this agent is shown in below:-

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

if status = Dirty then return Suck

else if location = A **then return** Right

else if location = B **then return** Left



Working of simple reflex agent

Q.2 a) List and explain performance measuring ways for problem solving. (5)

- There are variety of problem solving methods and algorithms available in AI. The performance of all these algorithms can be evaluated on the basis of following factors.
- The output of a problem-solving is either failure or a solution. We will evaluate an algorithm‘s performance in four ways:

I. **Completeness:**
   - If the algorithm is able to produce the solution if one exists then it satisfies completeness criteria.

II. **Optimality:**
   - If more than one way exists to derive the solution then the best one is selected.
   - Does the strategy find the optimal solution?
   - If the solution produced is the minimum cost solution, the algorithm is said to be optimal.

III. **Time complexity:**
   - Time taken to run a solution.
   - How long does it take to find a solution?
   - It depends on the time taken to generate the solution. It is number of nodes generated during the search.

IV. **Space complexity:**

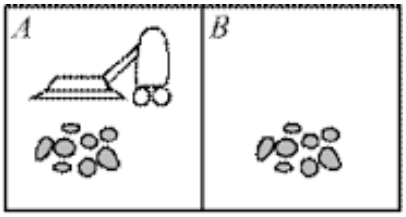
- Memory needed to perform the search.
- How much memory is needed to perform the search?
- Memory required to store the generated nodes while performing the search.

- Time and space complexity are always considered with respect to some measure of the problem difficulty. In theoretical computer science, the typical measure is the size of the state space graph, **|V | + |E|,**
    - where V is the set of vertices (nodes) of the graph and
    - E is the set of edges (links).

  This is appropriate when the graph is an explicit data structure that is input to the search program. (The map of Romania is an example of this.)
- In AI, the graph is often represented implicitly by the initial state, actions, and transition model and is frequently infinite. For these reasons, complexity is expressed in terms of three quantities:
    - **b**, the branching factor or maximum number of successors of any node;
    - **d**, the depth of the shallowest goal DEPTH node (i.e., the number of steps along the path from the root); and
    - **m**, the maximum length of any path in the state space. Time is often measured in terms of the number of nodes generated during the search, and space in terms of the maximum number of nodes stored in memory.
- For the most part, we describe time and space complexity for search on a tree; for a graph, the answer depends on how "redundant" the paths in the state space are. SEARCH COST To assess the effectiveness of a search algorithm, we can consider just the search cost— which typically depends on the time complexity but can also include a term for memory TOTAL COST usage—or we can use the total cost, which combines the search cost and the path cost of the solution found.
- For the problem of finding a route from Arad to Bucharest, the search cost is the amount of time taken by the search and the solution cost is the total length of the path in kilometers.
  Thus, to compute the total cost, we have to add milliseconds and kilometers.
- There is no "official exchange rate" between the two, but it might be reasonable in this case to convert kilometers into milliseconds by using an estimate of the car's average speed (because time is what the agent cares about). This enables the agent to find an optimal trade-off point at which further computation to find a shorter path becomes counterproductive.

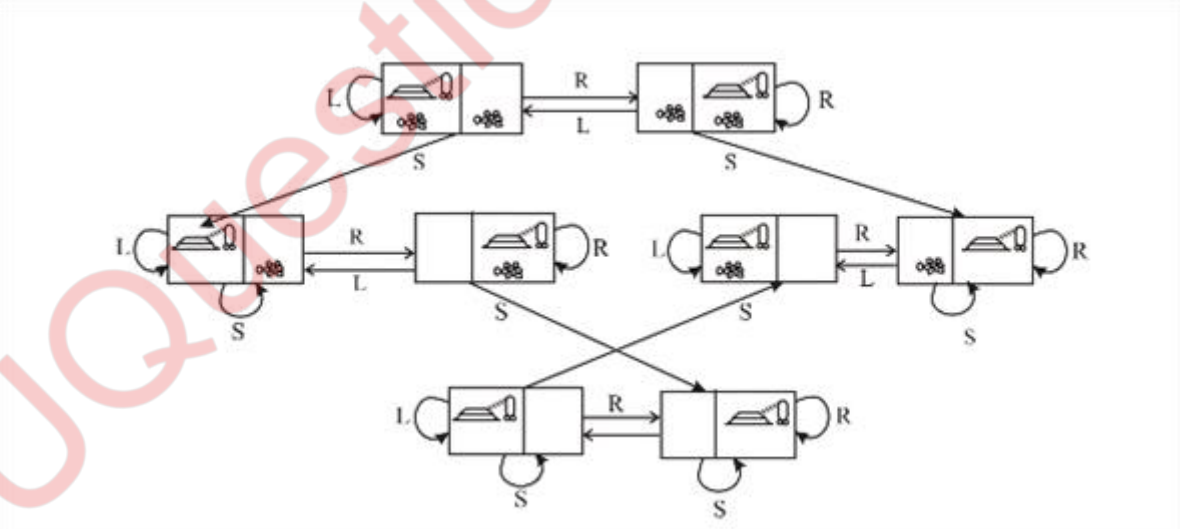
Q.2 b) Formulate the vacuum world problem. **(5)**

Consider a Vacuum cleaner world



Imagine that our intelligent agent is a robot vacuum cleaner.
Let's suppose that the world has just two rooms. The robot can be in either room and there can be dirt in zero, one or two rooms.

1. **States: -**
   In vacuum cleaner problem, state can be represented as [<block>, clean] or [<block>, dirty]. Hence there are total 8 states in the vacuum cleaner world.
2. **Initial state: -**
   Any state can be considered as initial state. For example, [A, dirty]
3. **Actions: -**
   The possible actions for the vacuum cleaner machine are left, right, absorb, and idle.
4. **Successor function: -**
   In fig indicating all possible states with actions and the next state.
5. **Goal state:-**
   The aim of the vacuum cleaner is to clean both the blocks. Hence the goal test if [A, Clean] and [B, Clean].
6. **Path Cost:-**
   Assuming that each action/ step costs 1 unit cost. The path cost is number of actions/ steps taken.

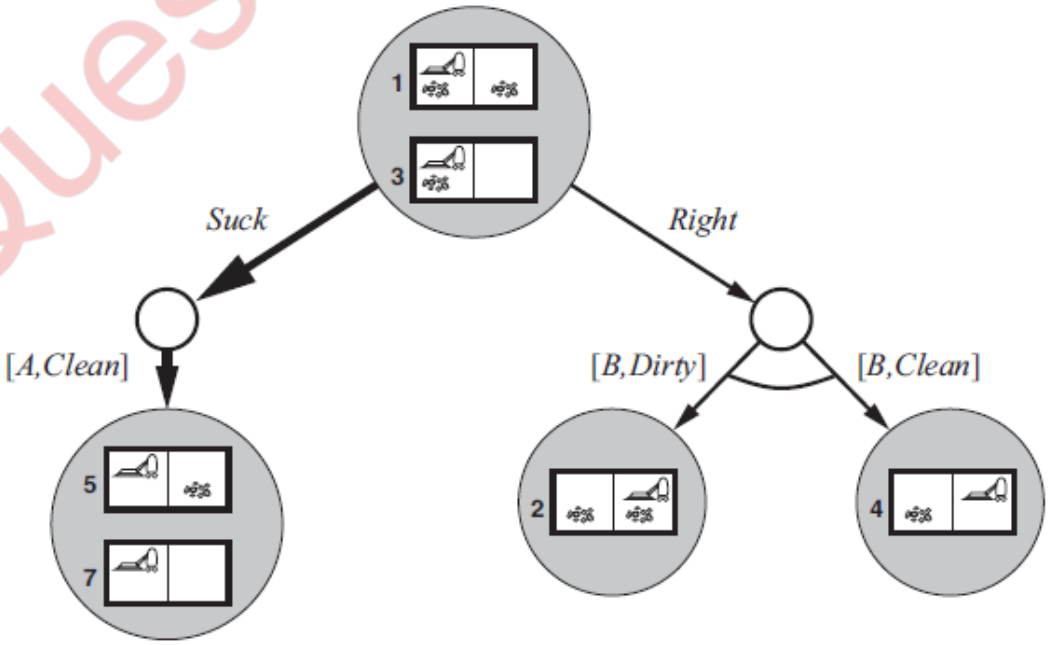


The state space for vacuum world

- **<u>Fully observable: -</u>**
- Search in belief state space, where the problem is fully observable!
  Solution is a sequence, even if the environment is non-deterministic!
  Suppose the underlying problem (P) is

$\{\text{Actions}_p, \text{Result}_p, \text{Goal} - \text{Test}_p, \text{Step} - \text{Cost}_p\}$

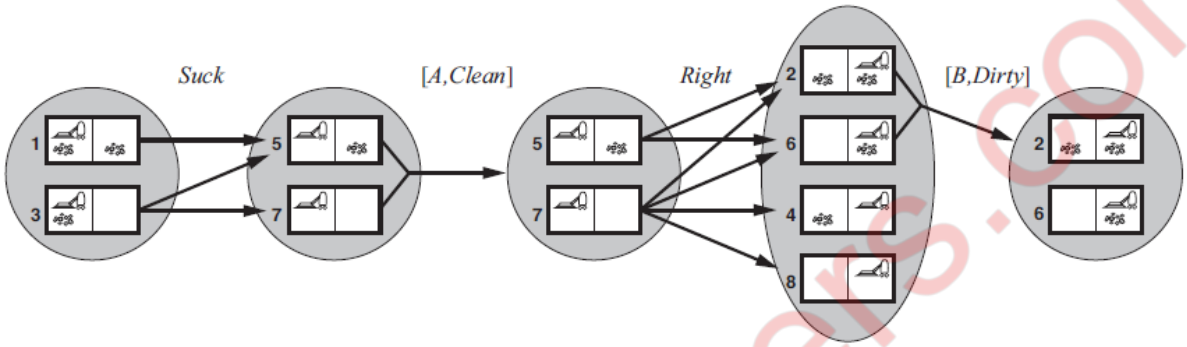
- What is the corresponding sensor-less problem
  States → Belief States: every possible set of physical states
  If N physical states, number of belief states can be 2^N
  Initial State: Typically the set of all states in P
- Actions: Consider {s1, s2}
- If Actions_p(s1) != Actions_p(s2) should we take the Union of both sets of actions or the Intersection?
- Union if all actions are legal, intersection if not
- If the environment is completely observable, 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.
- If the vacuum cleaner has no sensors, it doesn't know where it or the dirt is. In spite of this, it can still solve the problem. Here, states are knowledge states. States for the search: The power set of the world states 1-8.
- **Non-deterministic in action: -**
- Slippery vacuum world
- If at first you don't succeed try, try again
- We need to add label to some portion of a plan and use the label to refer to that portion – rather than repeating the sub plan → And-Or graphs with labels
- Plan: [Suck, L1: Right, if State == 5 then L1 else Suck]
- **And-Or solution: -**
- Given this problem formulation, we can use the And-Or search algorithm to come up with a plan to solve the problem
- Given [A, Dirty], Plan = {Suck, Right, if Bstate = {6} then Suck else []}



And-Or solution

- **Partially observable environments**

I. An agent in a partially observable environment must update belief state from percept
   - b' = Update(Predict(b, a), o)
   - So the agent is only looking at the current o (percept) not the entire history, as we considered earlier. This is recursive state estimation
   - Example: Kindergarten vacuum world



Partially Observable Environments

Q.2 c) Write the uniform cost search algorithm. Explain in short. (5)

II. Uniform Cost Search is an algorithm used to move around a directed weighted search space to go from a start node to one of the ending nodes with a minimum cumulative cost.

III. This search is an uninformed search algorithm, since it operates in a brute-force manner i.e. it does not take the state of the node or search space into consideration.

IV. It is used to find the path with the lowest cumulative cost in a weighted graph where nodes are expanded according to their cost of traversal from the root node. This is implemented using a priority queue where lower the cost higher is its priority.

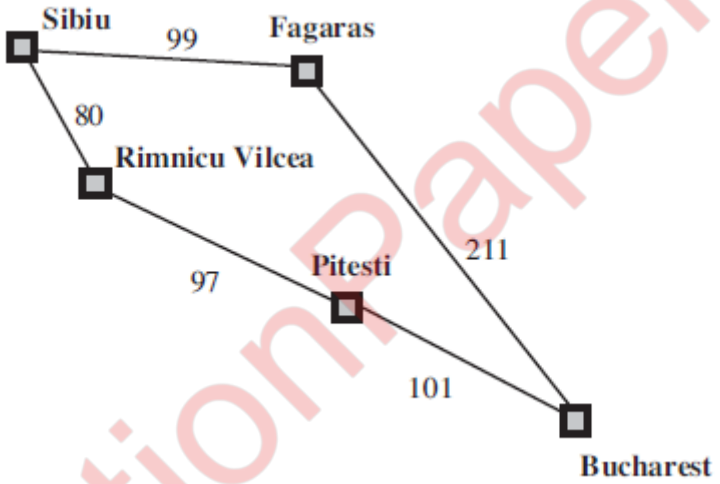
V. When all step costs are equal, breadth-first search is optimal because it always expands the shallowest unexpanded node. By a simple extension, we can find an algorithm that is optimal with any step-cost function. Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the lowest path cost g (n). This is done by storing the frontier as a priority queue ordered by g.

VI. In addition to the ordering of the queue by path cost, there are two other significant differences from breadth-first search. The first is that the goal test is applied to a node when it is selected for expansion. Rather than when it is first generated. The reason is that the first goal node that is generated may be on a suboptimal path. The second difference is that a test is added in case a better path is found to a node currently on the frontier.

VII. **Algorithm**

function UNIFORM-COST-SEARCH(problem) **returns** a solution, or failure
node ←a node with STATE = problem.INITIAL-STATE, PATH-COST = 0

frontier ←a priority queue ordered by PATH-COST, with node as the only element
explored ←an empty set
loop do
if EMPTY?(frontier) **then return** failure
node←POP(frontier) /* chooses the lowest-cost node in frontier */
if problem.GOAL-TEST(node.STATE) **then return** SOLUTION(node)
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**
frontier ←INSERT(child , frontier)
else if child .STATE is in frontier with higher PATH-COST **then**
replace that frontier node with child



Part of the Romania state space, selected to illustrate uniform-cost search.

VIII. The problem is to get from Sibiu to Bucharest. The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively. The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost 80 + 97=177.

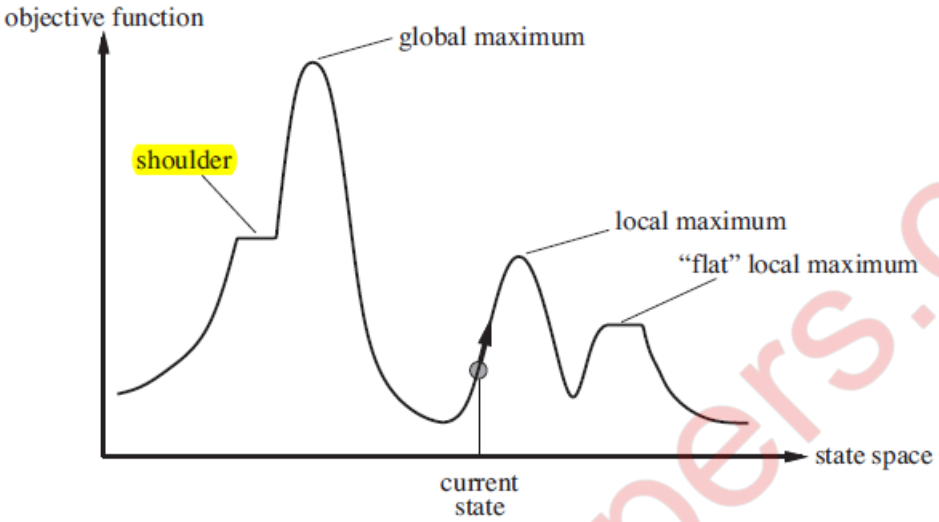
IX. The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost **99+211=310**. Now a goal node has been generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path to Bucharest with cost **80+97+101= 278**. Now the algorithm checks to see if this new path is better than the old one; it is, so the old one is discarded. Bucharest, now with g-cost 278, is selected for expansion and the solution is returned.

X. It is easy to see that uniform-cost search is optimal in general. Uniform-cost search does not care about the number of steps a path has, but only about their total cost. Uniform-cost search is guided by path costs rather than depths, so its complexity is not easily characterized in terms of b and d.

Q.2 d) With suitable diagram explain the following concepts. (5)

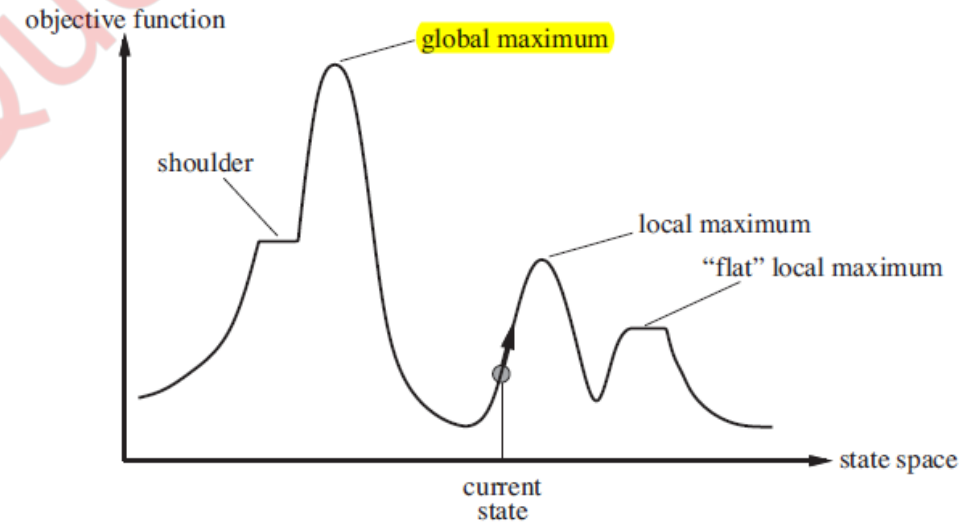
i. shoulder ii. Global maximum iii. Local maximum

i. Shoulder



- 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.
- It is a region having an edge upwards and it is also considered as one of the problems in hill climbing algorithms.
- The algorithm in Figure halts if it reaches a plateau where the best successor has the same value as the current state. Might it not be a good idea to keep going—to allow a sideways **move** in the hope that the plateau is really a shoulder, as shown in Figure? The answer is usually yes, but we must take care.
- 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.

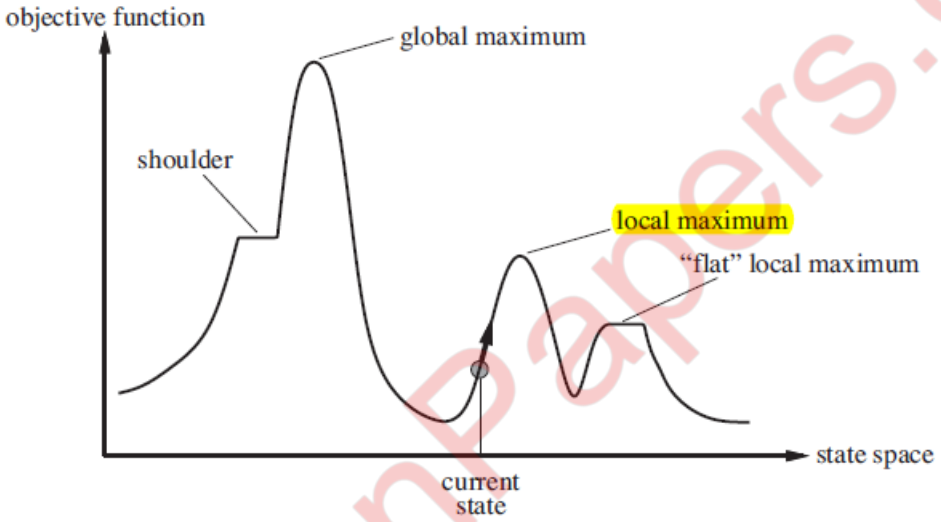
ii. Global maximum



- It is the highest state of the state space and has the highest value of cost function.

- It is a plateau that has an uphill edge.
- If elevation corresponds to an objective function, then the aim is to find the highest peak—a **global maximum**. (You can convert from one to the other just by inserting a minus sign.) Local search algorithms explore this landscape. A **complete** local search algorithm always finds a goal if one exists; an **optimal** algorithm always finds a global minimum/maximum.It is the best possible state in the state space diagram. This because at this state, objective function has highest value.

iii. <u>Local maximum</u>



- As visible from the diagram, it is the state which is slightly better than the neighbour states but it is always lower than the highest state.
- 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.
- It is a state which is better than its neighbouring state however there exists a state which is better than it (global maximum). This state is better because here the value of the objective function is higher than its neighbours.
- At a local maximum all neighbouring states have a values which is worse than the current state. Since hill-climbing uses a greedy approach, it will not move to the worse state and terminate itself. The process will end even though a better solution may exist.

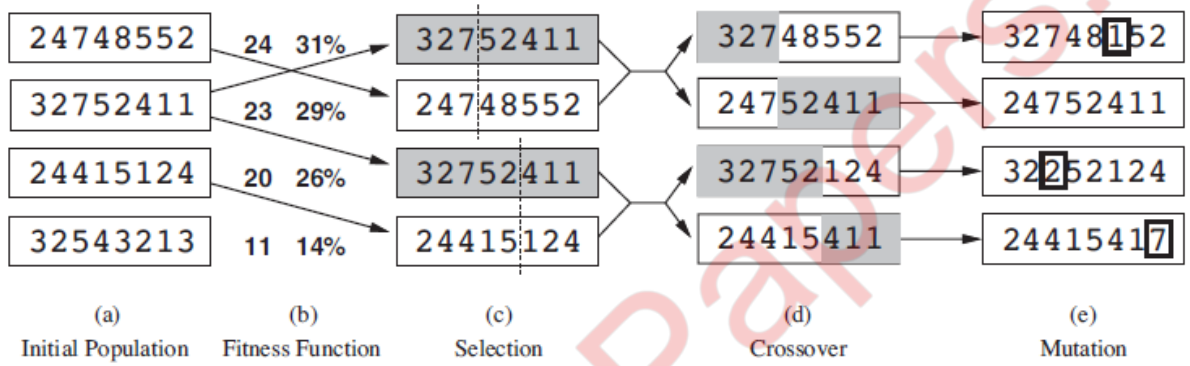
Q.2 e) How genetic algorithm works? (5)

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 f) Explain the working of AND-OR search tree. (5)

I. An **and–or tree** is a graphical representation of the reduction of problems (or goals) to conjunctions and disjunctions of sub problems (or sub goals).

II. In a deterministic environment, the only branching is introduced by the agent's own choices in each state. We call these nodes **OR nodes**. In the vacuum world, for example, at an OR node the agent chooses Left or Right or Suck. In a nondeterministic environment, branching is also introduced by the environment's choice of outcome for each action. We call these nodes **AND nodes**.

III. For example, the Suck action in state 1 leads to a state in the set {5, 7}, so the agent would need to find a plan for state 5 and for state 7. These two kinds of nodes alternate, leading to an AND–OR **tree** as illustrated in Figure

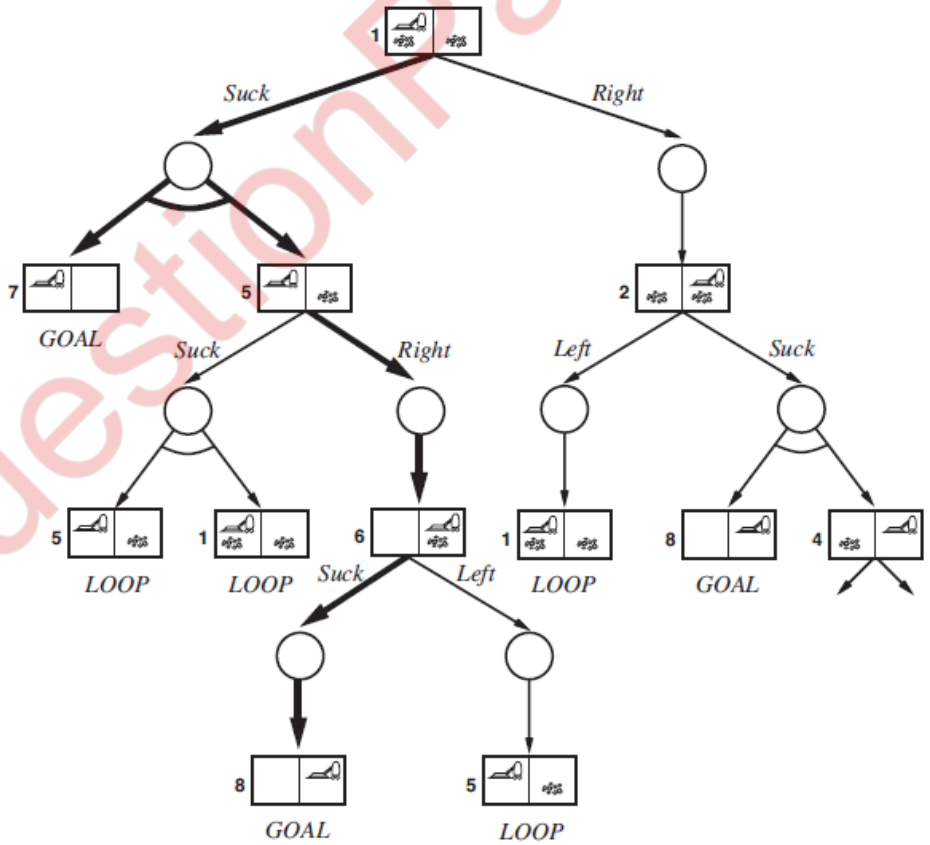
IV. A solution for an AND–OR search problem is a sub tree that (1) has a goal node at every leaf, (2) specifies one action at each of its OR nodes, and (3) includes every outcome branch at each of its AND nodes.

V. The solution is shown in bold lines in the figure; it corresponds to the plan given in Equation (The plan uses if–then–else notation to handle the AND branches, but when there are more than two branches at a node, it might be better to use a **case** construct.) Modifying the basic problem-solving agent to execute contingent solutions of this kind is straightforward. One may also consider a somewhat different agent design, in which the agent can act before it has found a guaranteed plan and deals with some contingencies only as they arise during execution.

VI. One key aspect of the algorithm is the way in which it deals with cycles, which often arise in nondeterministic problems (e.g., if an action sometimes has no effect or if an unintended effect can be corrected). If the current state is identical to a state on the path from the root, then it returns with failure. This doesn't mean that there is no solution from the current state; it simply means that if there is a noncyclic solution, it must be reachable from the earlier incarnation of the current state, so the new incarnation can be discarded.

VII. With this check, we ensure that the algorithm terminates in every finite state space, because every path must reach a goal, a dead end, or a repeated state. Notice that the algorithm does not check whether the current state is a repetition of a state on some other path from the root, which is important for efficiency.

VIII. AND–OR graphs can also be explored by breadth-first or best-first methods.



AND-OR search tree

Q.3 a) List and explain the elements used to define the game formally. (5)

I. Games have engaged the intellectual faculties of humans—sometimes to an alarming degree—for as long as civilization has existed. For AI researchers, the abstract nature of games makes them an appealing subject for study. The state of a game is easy to represent, and agents are usually restricted to a small number of actions whose outcomes are defined by precise rules.

II. We first consider games with two players, whom we call MAX and MIN for reasons that will soon become obvious. MAX moves first, and then they take turns moving until the game is over. At the end of the game, points are awarded to the winning player and penalties are given to the loser.

III. A game can be formally defined as a kind of search problem with the following elements:

1. **S0:** The initial state, which specifies how the game is set up at the start.
2. **PLAYER(s):** Defines which player has the move in a state.
3. **ACTIONS(s):** Returns the set of legal moves in a state.
4. **RESULT(s, a):** The transition model, which defines the result of a move.
5. **TERMINAL-TEST(s):** A terminal test, which is true when the game is over and false otherwise. States where the game has ended are called terminal states.
6. **UTILITY(s, p):** A utility function (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state s for a player p. In chess, the outcome is a win, loss, or draw, with values +1, 0, or 1/2. Some games have a wider variety of possible outcomes; the payoffs in backgammon range from 0 to +192. A zero-sum game is (confusingly) defined as one where the total payoff to all players is the same for every instance of the game. Chess is zero-sum because every game has payoff of either 0 + 1, 1 + 0 or 1/2+ 1/2. "Constant-sum" would have been a better term, but zero-sum is traditional and makes sense if you imagine each player is charged an entry fee of 1/2.

Q.3 b) Write the minimax algorithm. Explain in short. (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).

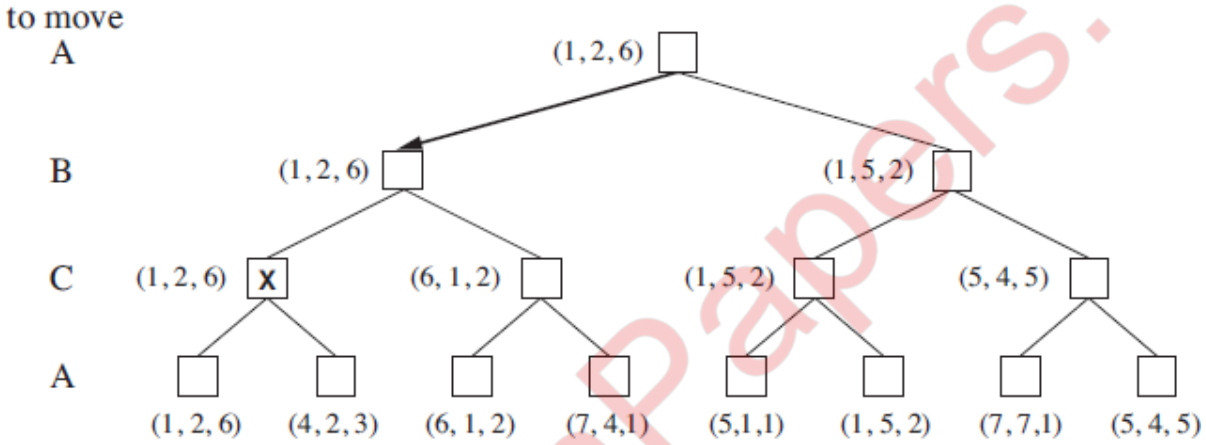
V. The space complexity is O (bm) for an algorithm that generates all actions at once, or O (m) for an algorithm that generates actions one at a time. For real games, of course, the time cost is totally impractical, but this algorithm serves as the basis for the mathematical analysis of games and for 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 c) Explain alpha-beta pruning with suitable example. (5)

I. Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.

II. As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent, but we can cut it to half. Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called **pruning**. This involves two threshold parameter Alpha and beta for future expansion, so it is called **alpha-beta pruning**. It is also called as **Alpha-Beta Algorithm**.

III. Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.

IV. The two-parameter can be defined as:

a. **Alpha:** The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is -∞.

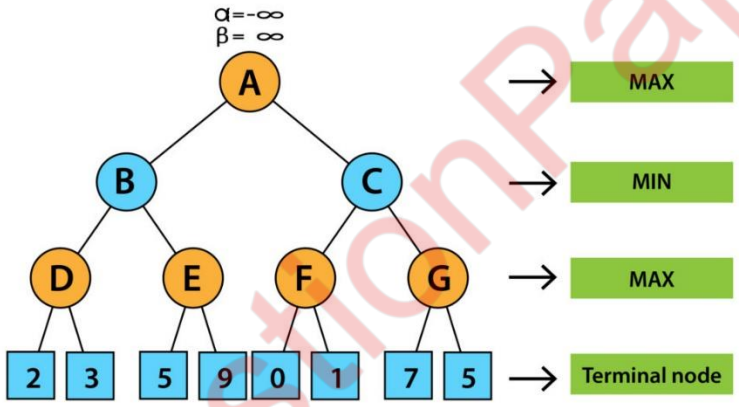
b. **Beta:** The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is +∞.

VI. The Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.

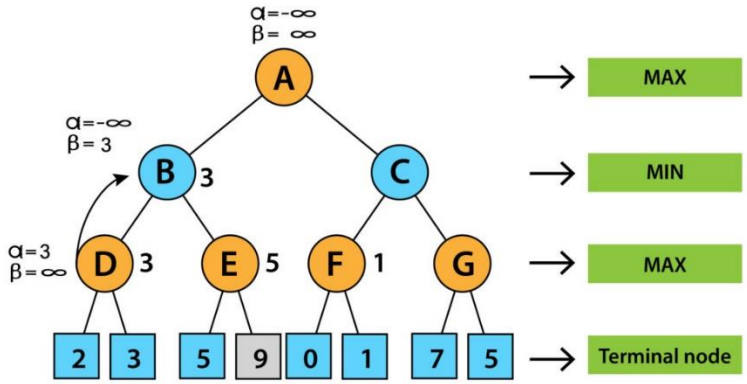
VII. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

VIII. Alpha–beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves.

- **<u>Example of alpha beta pruning</u>**
- We will first start with the initial move. We will initially define the alpha and beta values as the worst case i.e. α = -∞ and β= +∞. We will prune the node only when alpha becomes greater than or equal to beta.

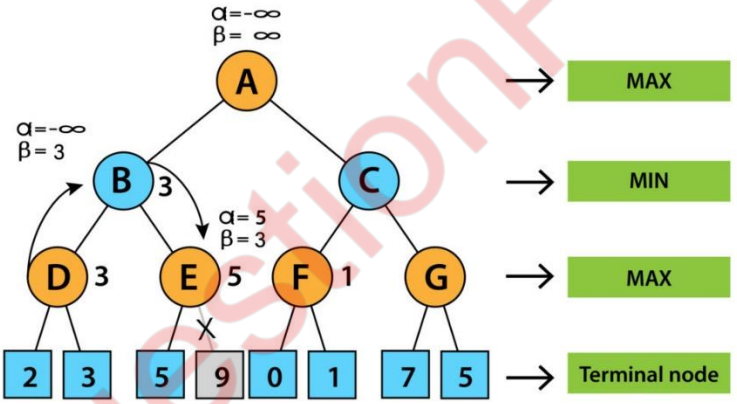


- Since the initial value of alpha is less than beta so we didn't prune it. Now it's turn for MAX. So, at node D, value of alpha will be calculated. The value of alpha at node D will be max (2, 3). So, value of alpha at node D will be 3.
- Now the next move will be on node B and its turn for MIN now. So, at node B, the value of alpha beta will be min (3, ∞). So, at node B values will be alpha= – ∞ and beta will be 3



- In the next step, algorithms traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.

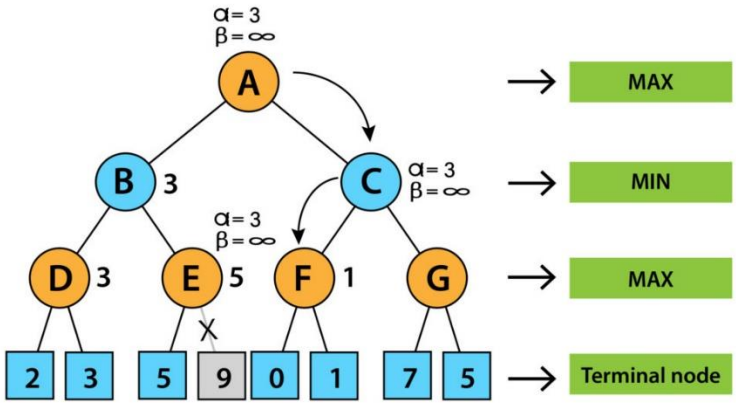
- Now it's turn for MAX. So, at node E we will look for MAX. The current value of alpha at E is – ∞ and it will be compared with 5. So, MAX (- ∞, 5) will be 5. So, at node E, alpha = 5, Beta = 5. Now as we can see that alpha is greater than beta which is satisfying the pruning condition so we can prune the right successor of node E and algorithm will not be traversed and the value at node E will be 5.



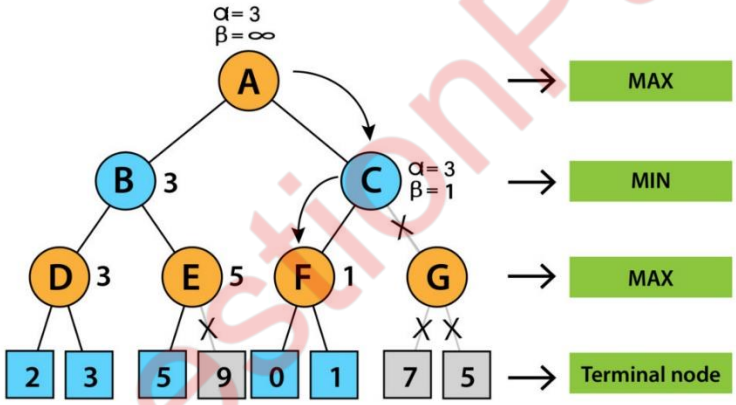
- In the next step the algorithm again comes to node A from node B. At node A alpha will be changed to maximum value as MAX (- ∞, 3). So now the value of alpha and beta at node A will be (3, + ∞) respectively and will be transferred to node C. These same values will be transferred to node F.

- At node F the value of alpha will be compared to the left branch which is 0. So, MAX (0, 3) will be 3 and then compared with the right child which is 1, and MAX (3,1) = 3

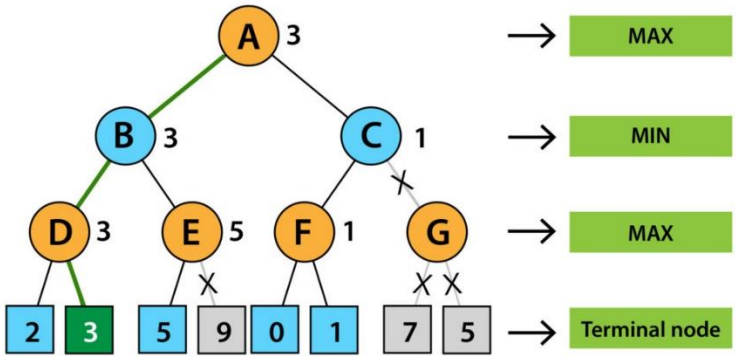
still α remains 3, but the node value of F will become 1.



- Now node F will return the node value 1 to C and will compare to beta value at C. Now its turn for MIN. So, MIN (+ ∞, 1) will be 1. Now at node C, α= 3, and β= 1 and alpha is greater than beta which again satisfies the pruning condition. So, the next successor of node C i.e. G will be pruned and the algorithm didn't compute the entire subtree G.



- Now, C will return the node value to A and the best value of A will be MAX (1, 3) will be 3.



- The above represented tree is the final tree which is showing the nodes which are computed and the nodes which are not computed. So, for this example the optimal value of the maximizer will be 3.

Q.3 d) Write the connectives used to form complex sentence of propositional logic.

Give Example of each. **(5)**

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 e) Explain the concept of knowledge base with example. (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 f) Write a short note on propositional theorem proving. (5)

- Reasoning by theorem proving is a weak method, compared to experts systems, because it does not make use of domain knowledge. This, on the other hand, may be a strength, if no domain heuristics are available (reasoning from first principles). Theorem proving is usually limited to sound reasoning.

- proving theorems is considered to require high intelligence
- if knowledge is represented by logic, theorem proving is reasoning
- theorem proving uses AI techniques, such as (heuristic) search
- (Study how people prove theorems. Differently!)

Theorem proving requires

- a logic (syntax)
- a set of axioms and inference rules
- a strategy on when how to search through the possible applications of the axioms and rules

I. Entailment can be done by **theorem proving**—applying rules of inference directly to the sentences in our knowledge base to construct a proof of the desired sentence without consulting models. If the number of models is large but the length of the proof is short, then theorem proving can be more efficient than model checking.

II. We will need some additional concepts related to entailment. The first concept is **logical equivalence**: two sentences α and β are logically equivalent if they are true in the same set of models. We write this as α ≡ β. For example, we can easily show (using truth tables) that P ∧ Q and Q ∧ Pare logically equivalent. These equivalences play much the same role in logic as arithmetic identities do in ordinary mathematics. An alternative definition of equivalence is as follows: any two sentences α and β are equivalent only if each of them entails the other:
α ≡ β if and only if α |= β and β |= α .

III. The second concept we will need is validity. A sentence is valid if it is true in all models. For example, the sentence P ∨ ¬P is valid. Valid sentences are also known as tautologies. They are necessarily true. Because the sentence True is true in all models, every valid sentence is logically equivalent to True. What good are valid sentences? From our definition of entailment, we can derive the deduction theorem, which was known to the ancient Greeks: For any sentences α and β, α |= β if and only if the sentence (α ⇒β) is valid.

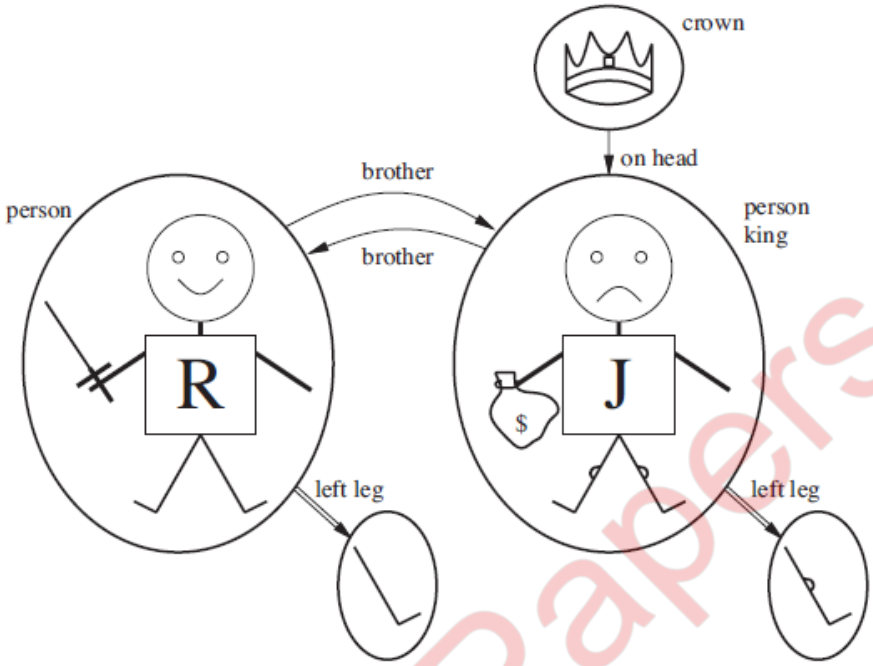
IV. The final concept we will need is **satisfiability**. A sentence is satisfiable if it is true in, or satisfied by, some model. For example, the knowledge base given earlier, (R1 ∧ R2 ∧R3 ∧ R4 ∧ R5), is satisfiable because there are three models in which it is true. Satisfiability can be checked by enumerating the possible models until one is found that satisfies the sentence. The problem of determining the satisfiability of sentences in propositional logic—the **SAT** problem—was the first problem proved to be NP-complete. Many problems in computer science are really satisfiability problems.

V. Validity and satisfiability are of course connected: α is valid if ¬α is unsatisfiable; contrapositively, α is satisfiable if ¬α is not valid.

VI. α |= β if and only if the sentence (α ∧ ¬β) is unsatisfiable
Proving β from α by checking the unsatisfiability of (α ∧ ¬β) corresponds exactly to the standard mathematical proof technique of reductio ad absurdum (literally, "reduction to an absurd thing"). It is also called proof by **refutation** or proof by **contradiction**. One assumes a sentence β to be false and shows that this leads to a contradiction with known axioms α. This contradiction is exactly what is meant by saying that the sentence (α ∧ ¬β) is unsatisfiable.

Q.4 a) Explain the following with example **(5)**

i. Atomic sentence **ii. Complex sentence**



A model containing five objects, two binary relations, three unary relations (indicated by labels on the objects), and one unary function, left-leg.

1. **Atomic sentence**
   - An atomic sentence (or atom for short) is formed from a predicate symbol optionally followed by a parenthesized list of terms, such as Brother (Richard, John).
   - This states, under the intended interpretation given earlier, that Richard the Lionheart is the brother of King John. Atomic sentences can have complex terms as arguments. Thus, Married (Father (Richard), Mother (John)) states that Richard the Lionheart's father is married to King John's mother (again, under a suitable interpretation).
   - An atomic sentence is true in a given model if the relation referred to by the predicate symbol holds among the objects referred to by the arguments.
   - **Atomic Sentence** = predicate (term 1,....., term n) or term1 = term2
   - An **atomic sentence** is formed from a predicate symbol followed by list of terms.
   - Examples:-
     - LargeThan(2,3) is false.
     - Brother_of(Mary,Pete) is false.
     - Married(Father(Richard),Mother(John)) could be true or false.

2. **Complex sentence**

- We can use **logical connectives** to construct more complex sentences, with the same syntax and semantics as in propositional calculus.
- Here are four sentences that are true in the model of Figure under our intended interpretation:
    - ¬Brother (LeftLeg(Richard), John)
    - Brother (Richard , John) ∧ Brother (John,Richard)
    - King(Richard) ∨ King(John)
    - ¬King(Richard) ⇒ King(John) .

Q.4 b) Explain universal Quantifier with 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) .

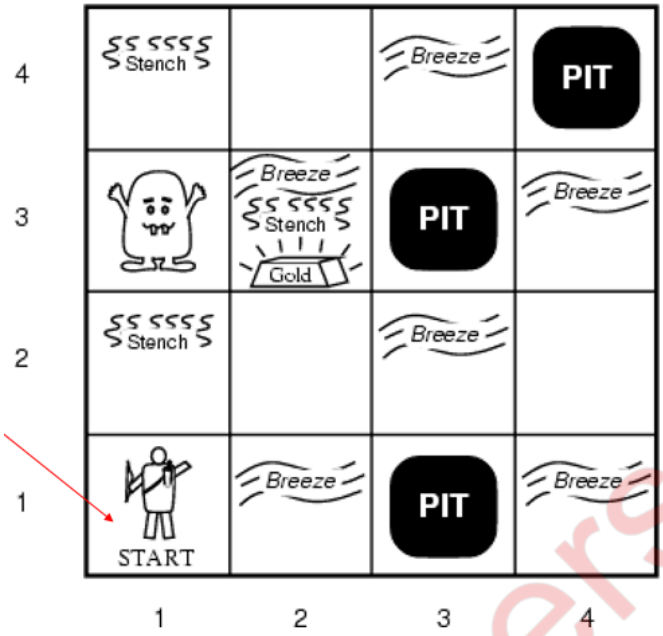
- ∃x is pronounced “There exists an x such that . . .” or “For some x . . .”

Q.4 c) Define the wumpus world problem in terms of first order logic. (5)

- Propositional logic can only represent facts about the world.
  First-order logic describes a world which consists of objects and properties (or predicates) of those objects.
  Among objects, various relations hold, e.g., Parent(Martin, Zac).
  A **function** is a relation in which there is only one value for a given input.
- **Examples**
    - Objects: people, houses, numbers, planets,...
    - Relations: parent, brother-of, greater-than,...
    - Properties: red, round, prime,...
    - Functions: father-of, one-more-than
- **Examples:**
    - "One plus one equals two."
    - "Squares neighbouring the Wumpus are smelly."
    - First-order logic is universal in the sense that it can express anything that can be programmed.

The Wumpus World environment

- The Wumpus computer game
- The agent explores a cave consisting of rooms connected by passageways.
- Lurking somewhere in the cave is the Wumpus, a beast that eats any agent that enters its room.
- Some rooms contain bottomless pits that trap any agent that wanders into the room.
- Occasionally, there is a heap of gold in a room.
- The goal is:
    - to collect the gold and
    - exit the world
    - without being eaten



I. Recall that the wumpus agent receives a percept vector with five elements. The corresponding first-order sentence stored in the knowledge base must include both the percept and the time at which it occurred; otherwise, the agent will get confused about when it saw what.

II. We use integers for time steps. A typical percept sentence would be Percept ([Stench, Breeze, Glitter, None, None], 5). Here, Percept is a binary predicate, and Stench and so on are constants placed in a list. The actions in the wumpus world can be represented by logical terms: Turn (Right), Turn (Left), Forward, Shoot, Grab, Climb.

III. To determine which is best, the agent program executes the query
ASKVARS(∃ a BestAction(a, 5)) ,
which returns a binding list such as {a/Grab}. The agent program can then return Grab as the action to take. The raw percept data implies certain facts about the current state.
For example:
∀ t, s, g, m, c Percept ([s, Breeze, g,m, c], t) ⇒ Breeze(t) ,
∀ t, s, b, m, c Percept ([s, b, Glitter,m, c], t) ⇒ Glitter (t) ,and so on.
These rules exhibit a trivial form of the reasoning process called perception

IV. Simple "reflex" behavior can also be implemented by quantified implication sentences. For example, we have ∀ t Glitter (t) ⇒ BestAction(Grab, t) .Given the percept and rules from the preceding paragraphs, this would yield the desired conclusion BestAction(Grab, 5)—that is, Grab is the right thing to do.

V. We have represented the agent's inputs and outputs; now it is time to represent the environment itself. Let us begin with objects. Obvious candidates are squares, pits, and the wumpus. We could name each square—Square1,2 and so on—but then the fact that Square1,2 and Square1,3 are adjacent would have to be an "extra" fact, and we would need one such fact for each pair of squares. It is better to use a complex term in which the row and column appear as integers; for example, we can simply use the list term [1, 2]. Adjacency of any two squares can be defined as

∀ x, y, a, b Adjacent ([x, y], [a, b]) ⇔
(x = a ∧ (y = b − 1 ∨ y = b + 1)) ∨ (y = b ∧ (x = a − 1 ∨ x = a + 1)).

VI. We could name each pit, but this would be inappropriate for a different reason: there is no reason to distinguish among pits.10 It is simpler to use a unary predicate Pit that is true of squares containing pits. Finally, since there is exactly one wumpus, a constant Wumpus is just as good as a unary predicate (and perhaps more dignified from the wumpus's viewpoint). The agent's location changes over time, so we write At(Agent, s, t) to mean that the agent is at square s at time t. We can fix the wumpus's location with ∀t At(Wumpus, [2, 2], t).We can then say that objects can only be at one location at a time:
∀ x, s1, s2, t At(x, s1, t) ∧ At(x, s2, t) ⇒ s1 = s2 .

VII. Given its current location, the agent can infer properties of the square from properties of its current percept. For example, if the agent is at a square and perceives a breeze, then that square is breezy:
∀ s, t At(Agent, s, t) ∧ Breeze(t) ⇒ Breezy(s) .

VIII. It is useful to know that a square is breezy because we know that the pits cannot move about. Notice that Breezy has no time argument. Having discovered which places are breezy (or smelly) and, very important, not breezy (or not smelly), the agent can deduce where the pits are (and where the wumpus is). In first-order logic we can quantify over time, so we need just one successor-state axiom for each predicate, rather than a different copy for each time step.

Q.4 d) Explain the following concepts **(5)**

i. Universal Instantiation ii. Existential Instantiation

i. <u>Universal instantiation</u>

- The rule of **Universal Instantiation** (**UI** for short) says that we can infer any sentence obtained by substituting a **ground term** (a term without variables) for the variable. To write out the inference rule formally, we use the notion of **substitutions.**
- This rule says that any substitution instance of a proposition function can be validly deduced from a universal proposition. A universal proposition is true only when it has only true substitution instances. This is the necessary and sufficient condition for any true universal proposition. Therefore any true substitution instance can be validly deduced from the respective universal proposition.
- Let SUBST (θ, α) denote the result of applying the substitution θ to the sentence α. Then the rule is written
  ∀v α
  SUBST ({v/g}, α)
- For any variable v and ground term g. For example, the three sentences given earlier are obtained with the substitutions {x/John}, {x/Richard}, and {x/Father (John)}.
- Use the UI rule in the following way:

- First, remove the universal quantifier.
- Next, replace the resulting free variable by a constant.

ii. <u>Existential Instantiation</u>

- In the rule for **Existential Instantiation**, the variable is replaced by a single new constant symbol.
- This rule is applicable when the proposition has existential quantifier and in this case any symbol ranging from a through w is used as a substitute for the individual variable x. We can infer the truth of any substitution instance from existential quantification because existential quantification is true only when there is at least one true substitution instance
- The formal statement is as follows: for any sentence α, variable v, and constant symbol k that does not appear elsewhere in the knowledge base,
  ∃v α
  SUBST ({v/k}, α)
- For example, from the sentence
  ∃ X Crown(x) ∧ OnHead(x, John)
  We can infer the sentence
  Crown (C1) ∧ OnHead (C1, John)
  As long as C1 does not appear elsewhere in the knowledge base. Basically, the existential
- Sentence says there is some object satisfying a condition, and applying the existential instantiation rule just gives a name to that object.
- Use the rule EI in a statement in the following way:
  - First, remove the existential quantifier.
  - Next, replace the resulting free variable with a constant

---

Q.4 e) write and explain a simple backward-chaining algorithm for first-order knowledge bases. (5)

I. Backward chaining is the same idea as forward chaining except that you start with requiring the learner to complete the last step of the task analysis. This means that you will perform all the preceding steps either for or with the learner and then begin to fade your prompts with the last step only.

II. Reinforcement is provided contingent upon the last step being completed. Once the learner is able to complete the last step independently, you will require the learner to complete the last two steps before receiving a reinforcer, and so on, until the learner is able to complete the entire chain independently before receiving access to a reinforcer.

III. Backward chaining uses the same basic approach as forward chaining but in reverse order. That is, you start with the last step in the chain rather than the first. The therapist can either prompt the learner through the entire sequence, without opportunities for independent responding, until he gets to the final step (and then

teach that step), or the therapist can initiate the teaching interaction by going straight to the last step.

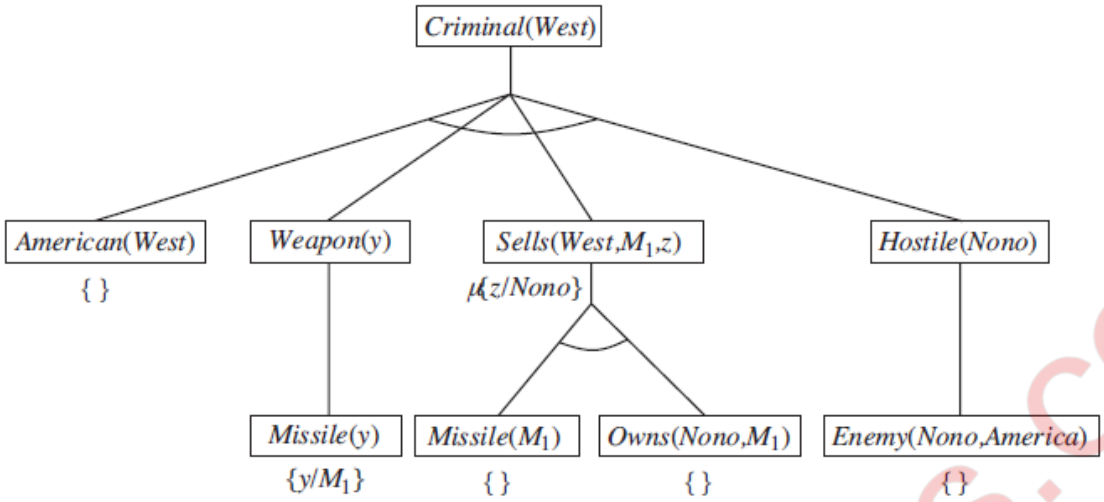
IV. Either way, when the last step occurs, the therapist uses prompting to help the learner perform the step correctly, reinforces correct responding with a powerful reinforcer, and then fades prompts across subsequent trials. When the last step is mastered, then each teaching interaction begins with the second-to-last step, and so on, until the first step in the chain is mastered, at which point the whole task analysis is mastered.

V. Backward chaining is a kind of AND/OR search—the OR part because the goal query can be proved by any rule in the knowledge base, and the AND part because all the conjuncts in the lhs of a clause must be proved.

VI. Backward chaining, as we have written it, is clearly a depth-first search algorithm. This means that its space requirements are linear in the size of the proof (neglecting, for now, the space required to accumulate the solutions). It also means that backward chaining (unlike forward chaining) suffers from problems with repeated states and incompleteness. We will discuss these problems and some potential solutions, but first we show how backward chaining is used in logic programming systems.

```
function FOL-BC-ASK(KB, query) returns a generator of substitutions
    return FOL-BC-OR(KB, query,{ })
generator FOL-BC-OR(KB, goal , θ) yields a substitution
    for each rule (lhs ⇒ rhs) in FETCH-RULES-FOR-GOAL(KB, goal ) do
        (lhs, rhs)←STANDARDIZE-VARIABLES((lhs, rhs))
        for each θ' in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal , θ)) do
            yield θ'
generator FOL-BC-AND(KB, goals, θ) yields a substitution
    if θ = failure then return
    else if LENGTH(goals) = 0 then yield θ
    else do
    first, rest ←FIRST(goals), REST(goals)
    for each θ' in FOL-BC-OR(KB, SUBST(θ, first), θ) do
    for each θ" in FOL-BC-AND(KB, rest , θ') do
    yield θ"
```

A simple backward-chaining algorithm for first-order knowledge bases



Tree constructed by backward chaining

- Proof tree constructed by backward chaining to prove that west is a criminal.
- The tree should be read depth first, left to right. To prove Criminal (West), we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. Bindings for each successful unification are shown next to the corresponding sub goal. Note that once one sub goal in a conjunction succeeds, its substitution is applied to subsequent sub goals. Thus, by the time FOL-BC-ASK gets to the last conjunct, originally Hostile (z), z is already bound to Nono.

Q.4 f) Explain the first order definite clause. (5)

I. First-order definite clauses closely resemble propositional definite clauses they are disjunctions of literals of which exactly one is positive. A definite clause either is atomic or is an implication whose antecedent is a conjunction of positive literals and whose consequent is a single positive literal. The following are first-order definite clauses:

- King(x) ∧ Greedy(x) ⇒ Evil(x)
- King(John)
- Greedy(y)

II. Unlike propositional literals, first-order literals can include variables, in which case those variables are assumed to be universally quantified. (Typically, we omit universal quantifiers when writing definite clauses.) Not every knowledge base can be converted into a set of definite clauses because of the single-positive-literal restriction, but many can. Consider the following problem:

– The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

III. We will prove that west is a criminal. First, we will represent these facts as first-order definite clauses. The next section shows how the forward-chaining algorithm solves the problem.

". . . it is a crime for an American to sell weapons to hostile nations":

American(x) ∧Weapon(y) ∧ Sells(x, y, z) ∧ Hostile(z) ⇒ Criminal (x)

"Nono . . . has some missiles." The sentence ∃ x Owns(Nono, x)∧Missile(x) is transformed into two definite clauses by Existential Instantiation, introducing a new constant M1:

Owns(Nono,M1)

Missile(M1)

"All of its missiles were sold to it by Colonel West":

Missile(x) ∧ Owns(Nono, x) ⇒ Sells(West, x, Nono) . (9.6)

We will also need to know that missiles are weapons:

Missile(x) ⇒ Weapon(x)

and we must know that an enemy of America counts as "hostile":

Enemy(x,America) ⇒ Hostile(x) . (9.8)

"West, who is American . . .":

American(West) . (9.9)

"The country Nono, an enemy of America . . .":

Enemy(Nono,America) . (9.10)

IV. This knowledge base contains no function symbols and is therefore an instance of the class of **Datalog** knowledge bases. Datalog is a language that is restricted to first-order definite clauses with no function symbols. Datalog gets its name because it can represent the type of statements typically made in relational databases. We will see that the absence of function symbols makes inference much easier.

Q.5 a) Write PDDL description of an air cargo transportation planning problem. (5)

- Planning Domain Definition Language (PDDL)
- Standard encoding language for "classical" planning tasks Components of a PDDL planning task:
  - Objects: Things in the world that interest us.
  - Predicates: Properties of objects that we are interested in; can be true or false.
  - Initial state: The state of the world that we start in.
  - Goal specification: Things that we want to be true.
  - Actions/Operators: Ways of changing the state of the world.

Init (At(C1, SFO) ∧ At(C2, JFK) ∧ At(P1, SFO) ∧ At(P2, JFK)
∧ Cargo(C1) ∧ Cargo(C2) ∧ Plane(P1) ∧ Plane(P2)
∧ Airport (JFK) ∧ Airport (SFO))
Goal (At(C1, JFK) ∧ At(C2, SFO))
Action(Load (c, p, a),

PRECOND: At(c, a) ∧ At(p, a) ∧ Cargo(c) ∧ Plane(p) ∧ Airport (a)
EFFECT: ¬ At(c, a) ∧ In(c, p))
Action(Unload(c, p, a),
PRECOND: In(c, p) ∧ At(p, a) ∧ Cargo(c) ∧ Plane(p) ∧ Airport (a)
EFFECT: At(c, a) ∧ ¬In(c, p))
Action(Fly(p, from, to),
PRECOND: At(p, from) ∧ Plane(p) ∧ Airport (from) ∧ Airport (to)
EFFECT: ¬ At(p, from) ∧ At(p, to))

A PDDL description of an air cargo transportation planning problem.

I. Figure shows an air cargo transport problem involving loading and unloading cargo and flying it from place to place. The problem can be defined with three actions: Load, Unload, and Fly.

II. The actions affect two predicates: In(c, p) means that cargo c is inside plane p, and At(x, a) means that object x (either plane or cargo) is at airport a. Note that some care must be taken to make sure the At predicates are maintained properly.

III. When a plane flies from one airport to another, all the cargo inside the plane goes with it. In first-order logic it would be easy to quantify over all objects that are inside the plane.

IV. But basic PDDL does not have a universal quantifier, so we need a different solution. The approach we use is to say that a piece of cargo ceases to be At anywhere when it is In a plane; the cargo only becomes At the new airport when it is unloaded. So At really means "available for use at a given location."

V. The following plan is a solution to the problem:

[Load (C1, P1, SFO), Fly (P1, SFO, JFK), Unload (C1, P1, JFK),
Load (C2, P2, JFK), Fly (P2, JFK, SFO), Unload (C2, P2, SFO)].

VI. Finally, there is the problem of spurious actions such as Fly (P1, JFK, JFK), which should be a no-op, but which has contradictory effects (according to the definition, the effect would include At(P1, JFK) ∧ ¬At(P1, JFK)). It is common to ignore such problems, because they seldom cause incorrect plans to be produced. The correct approach is to add inequality preconditions saying that the from and to airports must be different.

Q.5 b) Explain GRAPHPLAN algorithm. (5)

I. Planning graphs are an efficient way to create a representation of a planning problem that can be used to ⁄ Achieve better heuristic estimates ⁄ Directly construct plans

II. Planning graphs only work for propositional problems.

III. Planning graphs consists of a seq of levels that correspond to time steps in the plan. ⁄ Level 0 is the initial state. ⁄ Each level consists of a set of literals and a set of actions that represent what might be possible at that step in the plan ⁄ Might be is the key to

efficiency⁄Records only a restricted subset of possible negative interactions among actions.

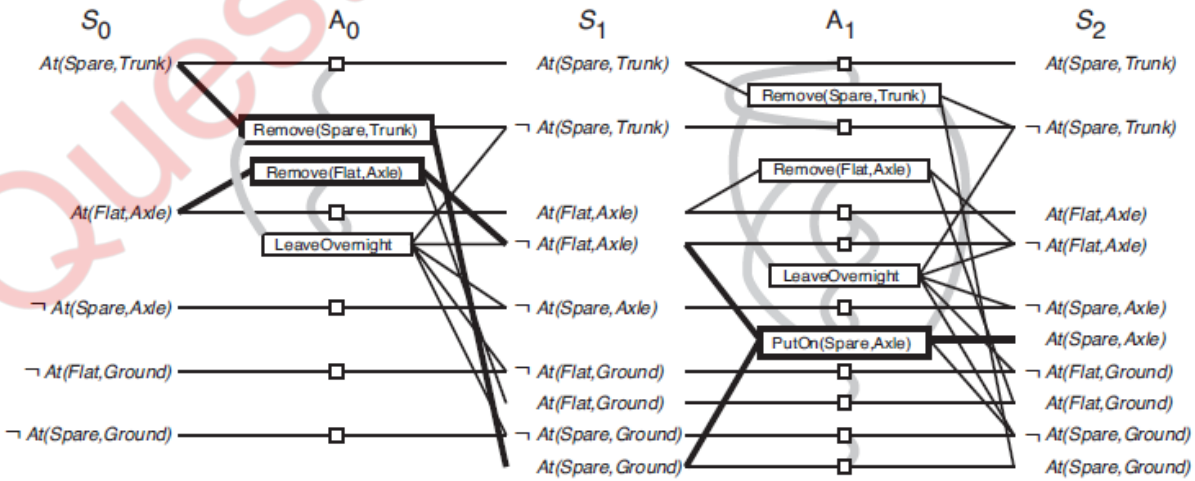
IV. Each level consists of

- Literals = all those that could be true at that time step, depending upon the actions executed at preceding time steps.
- Actions = all those actions that could have their preconditions satisfied at that time step, depending on which of the literals actually hold.

```
function GRAPHPLAN(problem) returns solution or failure
    graph ←INITIAL-PLANNING-GRAPH(problem)
    goals ←CONJUNCTS(problem.GOAL)
    nogoods ←an empty hash table
    for tl = 0 to∞do
      if goals all non-mutex in St of graph then
       solution ←EXTRACT-SOLUTION(graph, goals, NUMLEVELS(graph), nogoods)
       if solution _= failure then return solution
      if graph and nogoods have both leveled off then return failure
      graph←EXPAND-GRAPH(graph, problem)
```

The GRAPHPLAN algorithm. GRAPHPLAN calls EXPAND-GRAPH to add a level until either a solution is found by EXTRACT-SOLUTION, or no solution is possible.

V. The GRAPHPLAN algorithm repeatedly adds a level to a planning graph with EXPAND-GRAPH. Once all the goals show up as nonmutex in the graph, GRAPHPLAN calls EXTRACT-SOLUTION to search for a plan that solves the problem. If that fails, it expands another level and tries again, terminating with failure when there is no reason to go on.



- Initially the plan consist of 5 literals from the initial state and the CWA literals (S0).
- Add actions whose preconditions are satisfied by EXPAND-GRAPH (A0)
- Also add persistence actions and mutex relations.
- Add the effects at level S1
- Repeat until goal is in level Si

VI. EXPAND-GRAPH also looks for mutex relations

- Inconsistent effects:- E.g. Remove(Spare, Trunk) and LeaveOverNight due to At(Spare,Ground) and not At(Spare, Ground)
- Interference :- E.g. Remove(Flat, Axle) and LeaveOverNight At(Flat, Axle) as PRECOND and not At(Flat,Axle) as EFFECT
- Competing needs:- E.g. PutOn(Spare,Axle) and Remove(Flat, Axle) due to At(Flat.Axle) and not At(Flat, Axle)
- Inconsistent support:- E.g. in S2, At(Spare,Axle) and At(Flat,Axle)

Q.5 c) List various classical planning approaches. Explain any one. (5)

Currently the most popular and effective approaches to fully automated planning are:

• Translating to a Boolean satisfiability (SAT) problem

• Forward state-space search with carefully crafted heuristics

• Search using a planning graph

Classical planning as Boolean satisfiability

I. Here we show how to translate a PDDL description into a form that can be processed by SATPLAN. The translation is a series of straightforward steps:

II. Propositionalize the actions: replace each action schema with a set of ground actions formed by substituting constants for each of the variables. These ground actions are not part of the translation, but will be used in subsequent steps.

III. Define the initial state: assert F0 for every fluent F in the problem's initial state, and ¬F for every fluent not mentioned in the initial state.

IV. Propositionalize the goal: for every variable in the goal, replace the literals that contain the variable with a disjunction over constants. For example, the goal of having block A on another block, On (A, x) ∧ Block(x) in a world with objects A, B and C, would be replaced by the goal
(On (A, A) ∧ Block (A)) ∨ (On (A, B) ∧ Block (B)) ∨ (On (A, C) ∧ Block (C)).

V. Add successor-state axioms: For each fluent F, add an axiom of the form

$$F^{t+1} \Leftrightarrow ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNotF^t),$$

where ActionCausesF is a disjunction of all the ground actions that have F in their add list, and ActionCausesNotF is a disjunction of all the ground actions that have F in their delete list.

VI. Add precondition axioms: For each ground action A, add the axiom $A^t \Rightarrow PRE(A)^t$, that is, if an action is taken at time t, then the preconditions must have been true.

VII. Add action exclusion axioms: say that every action is distinct from every other action. The resulting translation is in the form that we can hand to SATPLAN to find a solution.

Q.5 d) Explain the following terms **(5)**

i. Circumscription **ii. Default logic**

i. <u>Circumscription</u>

- **Circumscription** is a non-monotonic logic created by John McCarthy to formalize the common sense assumption that things are as expected unless otherwise specified. Circumscription was later used by McCarthy in an attempt to solve the frame problem. To implement circumscription in its initial formulation, McCarthy augmented first-order logic to allow the minimization of the extension of some predicates, where the extension of a predicate is the set of tuples of values the predicate is true on. This minimization is similar to the closed-world assumption that what is not known to be true is false.
- The original problem considered by McCarthy was that of missionaries and cannibals: there are three missionaries and three cannibals on one bank of a river; they have to cross the river using a boat that can only take two, with the additional constraint that cannibals must never outnumber the missionaries on either bank (as otherwise the missionaries would be killed and, presumably, eaten).
- Bird(x) ∧¬Abnormal 1(x) ⇒ Flies(x).

  If we say that Abnormal 1 is to be **circumscribed**, a circumscriptive reasoner is entitled to assume ¬Abnormal 1(x) unless Abnormal 1(x) is known to be true. This allows the conclusion Flies(Tweety) to be drawn from the premise Bird(Tweety), but the conclusion no longer holds if Abnormal 1(Tweety) is asserted.
- Circumscription can be viewed as an example of a **model preference** logic. In such logics, a sentence is entailed (with default status) if it is true in all preferred models of the KB, as opposed to the requirement of truth in all models in classical logic.
- The standard example for which multiple inheritance is problematic is called the "Nixon diamond." It arises from the observation that Richard Nixon was both a Quaker (and hence by default a pacifist) and a Republican (and hence by default not a pacifist). We can write this as follows:
  - Republican(Nixon) ∧ Quaker(Nixon)
  - Republican(x) ∧¬Abnormal 2(x) ⇒ ¬Pacifist (x)
  - Quaker(x) ∧ ¬Abnormal 3(x) ⇒ Pacifist (x)
- If we circumscribe Abnormal 2 and Abnormal 3, there are two preferred models: one in which Abnormal 2(Nixon) and Pacifist (Nixon) hold and one in which Abnormal 3(Nixon) and ¬Pacifist(Nixon) hold. Thus, the circumscriptive reasoner remains properly agnostic as to whether Nixon was a pacifist. If we wish, in addition, to assert that religious beliefs take precedence over political beliefs, we can use a formalism called **prioritized circumscription** to give preference to models where Abnormal 3 is minimized.

ii. Default logic

- **Default logic** is a non-monotonic logic proposed by Raymond Reiter to formalize reasoning with default assumptions.
- Default logic can express facts like "by default, something is true"; by contrast, standard logic can only express that something is true or that something is false. This is a problem because reasoning often involves facts that are true in the majority of cases but not always. A classical example is: "birds typically fly". This rule can be expressed in standard logic either by "all birds fly", which is inconsistent with the fact that penguins do not fly, or by "all birds that are not penguins and not ostriches and ... fly", which requires all exceptions to the rule to be specified. Default logic aims at formalizing inference rules like this one without explicitly mentioning all their exceptions.
- **Default logic** is a formalism in which **default rules** can be written to generate contingent, nonmonotonic conclusions. A default rule looks like this:

  Bird(x) : Flies(x)/Flies(x)
- This rule means that if Bird(x) is true, and if Flies(x) is consistent with the knowledge base, then Flies(x) may be concluded by default. In general, a default rule has the form

  P : J1, . . . , Jn/C

  where P is called the prerequisite, C is the conclusion, and Ji are the justifications—if any one of them can be proven false, then the conclusion cannot be drawn. Any variable that appears in Ji or C must also appear in P.
- The Nixon-diamond example can be represented in default logic with one fact and two default rules:
  - Republican(Nixon) ∧ Quaker(Nixon)
  - Republican(x) : ¬Pacifist (x)/¬Pacifist (x)
  - Quaker(x) : Pacifist (x)/Pacifist (x)
- To interpret what the default rules mean, we define the notion of an **extension** of a default theory to be a maximal set of consequences of the theory. That is, an extension S consists of the original known facts and a set of conclusions from the default rules, such that no additional conclusions can be drawn from S and the justifications of every default conclusion in S are consistent with S

Q.5 e) Write a short note on description logics. (5)

I. The syntax of first-order logic is designed to make it easy to say things about objects. Description logics are notations that are designed to make it easier to describe definitions and properties of categories.

II. Description logic systems evolved from semantic networks in response to pressure to formalize what the networks mean while retaining the emphasis on taxonomic structure as an organizing principle.

III. The principal inference tasks for description logics are subsumption (checking if one category is a subset of another by comparing their definitions) and classification (checking whether an object belongs to a category). Some systems also include consistency of a category definition—whether the membership criteria are logically satisfiable.

Concept → **Thing** | ConceptName
| **And**(Concept , . . .)
| **All**(RoleName, Concept)
| **AtLeast**(Integer, RoleName)
| **AtMost**(Integer, RoleName)
| **Fills**(RoleName , IndividualName, . . .)
| **SameAs**(Path, Path)
| **OneOf**(IndividualName, . . .)
Path → [RoleName, . . .]

The syntax of descriptions in a subset of the CLASSIC language

IV. The CLASSIC language (Borgida et al., 1989) is a typical description logic.
For example, to say that bachelors are unmarried adult males we would write

Bachelor = And(Unmarried, Adult ,Male) .

The equivalent in first-order logic would be

Bachelor (x) ⇔ Unmarried(x) ∧ Adult(x) ∧ Male(x).

V. Any description in CLASSIC can be translated into an equivalent first-order sentence, but some descriptions are more straightforward in CLASSIC.
For example, to describe the set of men with at least three sons who are all unemployed and married to doctors, and at most two daughters who are all professors in physics or math departments, we would use
And(Man, AtLeast(3, Son), AtMost(2, Daughter),
All(Son, And(Unemployed,Married, All(Spouse, Doctor))),
All(Daughter , And(Professor , Fills(Department , Physics,Math)))) .

Q.5 f) Explain semantic network with 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.

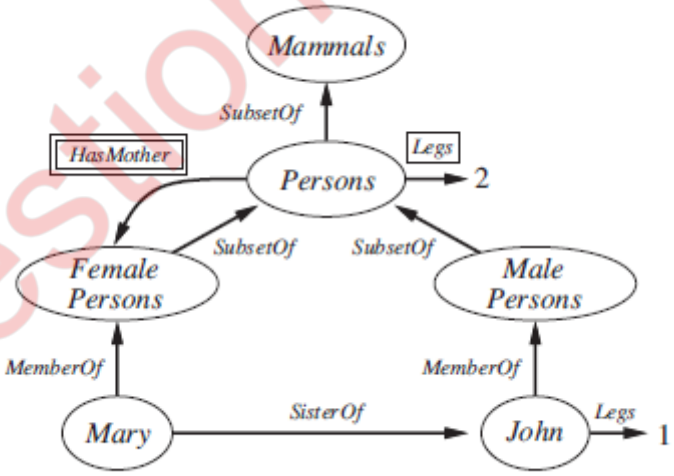
II. A **semantic network**, or **frame network** is a knowledge base that represents semantic relations between concepts in a network. It is a directed or undirected graph consisting of vertices, which represent concepts, and edges, which represent semantic relations between concepts, mapping or

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