GHOUSIA INSTITUTE OF TECHNOLOGY FOR WOMEN

Near Dairy Circle, Hosur Road, Bengaluru ,Karnataka 560029

Affiliated to VTU., Belagavi, Recognized by Government of Karnataka & A.I.C.T.E., New Delhi

ARTIFICIAL INTELLIGENCE

As per 2022 Scheme Syllabus Prescribed by V.T.U.

For

COMPUTER SCIENCE & ENGINEERING/INFORMATION SCIENCE & ENGINEERING/ROBOTICS & ARTIFICIAL INTELLIGENCE/CYBER SECURITY/ARTIFICIAL INTELLIGENCE & MACHINE LEARNING

(Bachelor of Engineering)

Dr.NAVEED M.Tech., PhD.

Assistant Professor

Department of Computer Science & Engineering

ARTIFICIAL INTELLIGENCE		Semester	V
Course Code	**BCS515B**	CIE Marks	50
Teaching Hours/Week (L: T:P: S)	3:0:0:0	SEE Marks	50
Total Hours of Pedagogy	40	Total Marks	100
Credits	03	Exam Hours	3
Examination type (SEE)	Theory		

Course objectives:

- Learn the basic principles and theories underlying artificial intelligence, including machine learning, neural networks, natural language processing, and robotics.
- Apply AI techniques to solve real-world problems, including search algorithms, optimization, and decision-making processes.
- Understand the ethical, legal, and societal implications of AI, including topics such as bias, fairness, accountability, and the impact of AI on the workforce and privacy.

Teaching-Learning Process (General Instructions)

These are sample Strategies, which teachers can use to accelerate the attainment of the various course outcomes.

1. Use of Video/Animation to explain functioning of various concepts.
2. Encourage collaborative (Group Learning) Learning in the class.
3. Discuss application of every concept to solve the real-world problems.
4. Demonstrate ways to solve the same problem and encourage the students to come up with their own creative solutions.

Module-1

Introduction: What Is AI? , The State of The Art.

Intelligent Agents: Agents and environment, Concept of Rationality, The nature of environment, The structure of agents.

Chapter 1 - 1.1, 1.4

Chapter 2 - 2.1, 2.2, 2.3, 2.4

Module-2

Problem-solving: Problem-solving agents, Example problems, Searching for Solutions Uninformed Search Strategies

Chapter 3 - 3.1, 3.2, 3.3, 3.4

Module-3

Problem-solving: Informed Search Strategies, Heuristic functions

Logical Agents: Knowledge–based agents, The Wumpus world, Logic, Propositional logic, Reasoning patterns in Propositional Logic

Chapter 3 - 3.5, 7.6

Chapter 7 - 7.1, 7.2, 7.3, 7.4

Module-4

First Order Logic: Representation Revisited, Syntax and Semantics of First Order logic, Using First Order logic, Knowledge Engineering In First-Order Logic

Inference in First Order Logic: Propositional Versus First Order Inference, Unification, Forward Chaining

Chapter 8- 8.1, 8.2, 8.3, 8.4

Chapter 9- 9.1, 9.2, 9.3

Module-5
Inference in First Order Logic: Backward Chaining, Resolution **Classical Planning:** Definition of Classical Planning, Algorithms for Planning as State-Space Search, Planning Graphs **Chapter 9-9.4, 9.5** **Chapter 10- 10.1,10.2,10.3**

Course outcomes (Course Skill Set)

At the end of the course, the student will be able to:

1. Explain the architecture and components of intelligent agents, including their interaction with the AI environment.
2. Apply problem-solving agents and various search strategies to solve a given problem.
3. Illustrate logical reasoning and knowledge representation using propositional and first-order logic.
4. Demonstrate proficiency in representing knowledge and solving problems using first-order logic.
5. Describe classical planning in the context of artificial intelligence, including its goals, constraints, and applications in problem-solving.

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together.

Continuous Internal Evaluation:

- For the Assignment component of the CIE, there are 25 marks and for the Internal Assessment Test component, there are 25 marks.
- The first test will be administered after 40-50% of the syllabus has been covered, and the second test will be administered after 85-90% of the syllabus has been covered
- Any two assignment methods mentioned in the 22OB2.4, if an assignment is project-based then only one assignment for the course shall be planned. The teacher should not conduct two assignments at the end of the semester if two assignments are planned.
- For the course, CIE marks will be based on a scaled-down sum of two tests and other methods of assessment.

Internal Assessment Test question paper is designed to attain the different levels of Bloom's taxonomy as per the outcome defined for the course.

Semester-End Examination:

Theory SEE will be conducted by University as per the scheduled timetable, with common question papers for the course (**duration 03 hours).**

1. The question paper will have ten questions. Each question is set for 20 marks.
2. There will be 2 questions from each module. Each of the two questions under a module (with

Suggested Learning Resources:

Text Book

Stuart J. Russell and Peter Norvig, Artificial Intelligence, 3rd Edition, Pearson, 2015

Reference Books

1. Elaine Rich, Kevin Knight, Artificial Intelligence, 3rd edition, Tata McGraw Hill, 2013
2. George F Lugar, Artificial Intelligence Structure and strategies for complex, Pearson Education, 5th Edition, 2011
3. Nils J. Nilsson, Principles of Artificial Intelligence, Elsevier, 1980
4. Saroj Kaushik, Artificial Intelligence, Cengage learning, 2014

Web links and Video Lectures (e-Resources):

1. https://www.kdnuggets.com/2019/11/10-free-must-read-books-ai.html
2. https://www.udacity.com/course/knowledge-based-ai-cognitive-systems--ud409
3. https://nptel.ac.in/courses/106/105/106105077/

Activity Based Learning (Suggested Activities in Class)/ Practical Based learning

1. Using OpenAI tool, develop a chatbot (25 marks)

MODULE-01 INTRODUCTION & AI AGENTS

Syllabus: Introduction: What Is AI? , The State of The Art. Intelligent Agents: Agents and environment, Concept of Rationality, The nature of environment, The structure of agents.

1.1 What Is AI?, The State of the Art:

Artificial Intelligence (AI) can be defined in many ways depending on what people expect from intelligent machines. Some definitions focus on machines that think, while others focus on machines that act. Similarly, some definitions aim for human-like performance, while others focus on ideal, rational performance. To organize these views, AI is commonly described using four different approaches:

1. Acting Humanly
2. Thinking Humanly
3. Thinking Rationally and
4. Acting Rationally.

1.1.1 Acting Humanly:

The first approach, acting humanly, is based on the Turing Test, proposed by Alan Turing in 1950. In this test, a human interacts with both a computer and another human through written text. If the human cannot reliably tell which one is the machine, then the machine is considered intelligent. To pass such a test, the machine needs to understand and respond in natural language, store and retrieve knowledge, reason logically, and learn from past interactions.
In the extended version, called the Total Turing Test, the machine must also perceive objects (using computer vision) and move or interact physically (using robotics).
For example, a chatbot that can carry on a meaningful conversation in English, and a robot that can see and move objects, would be considered strong candidates to pass the Turing Test.
AI is also behind speech recognition systems. When you call an airline or use a virtual assistant like Siri, you can talk to the system in plain English, and it responds meaningfully. This shows the system's ability to understand and use human language, which is part of acting humanly, as required by the Turing Test.

1.1.2 Thinking Humanly:

The second approach, thinking humanly, aims to design machines that think the way humans do. To do this, we must first understand how humans think. This is done through

i. Introspection (Observing Our Own Thoughts)
ii. Psychological Experiments (Observing Others) and
iii. Brain Imaging (Seeing Brain Activity While Thinking).

Once a theory of human thinking is formed, it can be converted into a computer program. If the program behaves similarly to a human in problem-solving or decision-making, it may reflect human-like thinking.
Imagine an AI system designed to solve math problems like a human student. Instead of just giving the final answer instantly, the AI tries to think step-by-step, just like a person would. It

breaks the problem into smaller parts, remembers past mistakes, and even takes longer if the question is more difficult — just like a real student. The AI was trained using data from how real students solve problems, including their thought patterns and errors. This is an example of thinking humanly, because the AI isn't just getting the right answer, it's trying to copy how humans actually think and reason through a problem.

This approach contributes to the field of cognitive science, which combines AI and psychology to build models that simulate the human mind.

1.1.3 Thinking Rationally:

Rational means making smart and sensible decisions by thinking clearly and using the information you have. A rational person or machine chooses the option that gives the best result or solves the problem most effectively.

For example, if you have ₹100 and you're hungry, choosing a filling meal instead of just chips is a rational decision because it satisfies your need better. In the same way, a rational AI uses data and logic to make the best possible choice to reach its goal.

The third approach, thinking rationally, is based on logic and reasoning. The goal is to create machines that always make logically correct decisions. Although this approach seems appealing, it faces challenges. Real-world knowledge is often uncertain or incomplete, making it difficult to express everything in strict logical terms. Also, reasoning through logic can become computationally expensive and slow for complex problems. Despite these issues, AI systems based on logical reasoning, like theorem provers and expert systems, are still important in the field.

For example imagine a delivery robot that needs to choose between two routes to deliver food. Route A is a shortcut but might be blocked due to construction, while Route B is longer but safe and reliable. The robot uses its sensors, GPS, and traffic data to evaluate both options. It finds that Route A has a high chance of delay, while Route B guarantees a successful delivery with a small time loss. Thinking rationally, the robot chooses Route B because it offers the best chance of completing the task successfully. This is an example of rational thinking in AI — using available information to make the smartest decision to achieve a goal.

In the world of games, AI has made history. IBM's Deep Blue beat the world chess champion Garry Kasparov, showing that AI can think and plan better than even the best human players in some cases. This fits into the thinking rationally category, where the system uses logic and strategy to make winning moves.

1.1.4 Acting Rationally:

The most widely accepted and modern approach is acting rationally. This focuses on building rational agents that do the best possible thing to achieve their goals, based on what they know and what they can perceive. Rationality does not mean perfection. A rational agent does not always make the perfect choice, but rather the best one based on the available information.

For example, if a self-driving car encounters heavy traffic or unpredictable road conditions, it must still decide the safest and most efficient action, even without knowing the future. Rational agents use knowledge representation, reasoning, learning, and planning to make intelligent decisions. This approach is considered practical and flexible, as it does not depend on mimicking human behavior but instead focuses on achieving desired outcomes intelligently.

Today, Artificial Intelligence (AI) is being used in many real-life areas, and it's growing rapidly. For example, self-driving cars like STANLEY and BOSS can drive themselves by using cameras, sensors, and smart decision-making software. These cars don't just follow rules—they act rationally by sensing the environment and choosing the best and safest actions to reach their

destination, just like a skilled human driver would. This reflects the acting rationally approach to AI.

In space, NASA uses AI for autonomous planning and scheduling. The "Remote Agent" controlled spacecraft activities millions of miles from Earth, planning its own tasks and fixing problems as they happened. This is a clear example of acting rationally, where the AI plans actions based on goals and current situations without human help.

During the Gulf War, the U.S. used AI to manage logistics planning—like moving tens of thousands of troops, vehicles, and supplies efficiently. AI created plans in hours that would have taken humans weeks, showing how AI can act rationally by solving large, complex problems faster than people.

At home, robots like Roomba use AI to clean floors without needing instructions. They sense obstacles and navigate rooms intelligently. This is another example of acting rationally, where the robot takes in information and makes decisions on its own.

In summary, from all these examples, AI systems are not doing magic—they're using science, logic, learning, and engineering to behave intelligently, either by thinking like humans, acting like humans, thinking rationally, or acting rationally, as described in the four main definitions of AI. Artificial intelligence can be approached in different ways depending on whether the focus is on thinking or acting, and whether the goal is to match human behavior or perform rationally.

The four approaches—acting humanly, thinking humanly, thinking rationally, and acting rationally—each offer different insights and methods. Among these, the rational agent approach is currently the most widely used because it provides a solid, scientific foundation for building intelligent systems that can function effectively in real-world environments.

Approach	Focus	Based on	Example
Acting Humanly	Human behavior	Turing Test	Chatbot like a human
Thinking Humanly	Human thought	Psychology, brain scans	Problem solver that mimics humans
Thinking Rationally	Logical thinking	Rules & logic	Theorem prover
Acting Rationally	Best actions	Rational decisions	Self-driving car

Foundations of AI:

Philosophy laid the groundwork for AI by exploring logic, knowledge, and reasoning.Early mechanical calculators showed how machines could simulate thought. Modern AI thrives on data—more data often means better performance, even with simpler algorithms. Learning from data has reduced the need for manual programming, making AI more powerful and widespread.

Philosophy's Role in AI:

Philosophers asked big questions that later became central to AI, such as:

- Can logic and rules be used to reach correct conclusions?
- How does the mind emerge from the physical brain?
- Where does knowledge come from?
- How does knowledge turn into action?

Aristotle (384–322 B.C.) was one of the first to try defining rules for logical reasoning. His system of syllogisms (like "All humans are mortal; Socrates is human; therefore, Socrates is mortal") showed how conclusions could be drawn mechanically from premises.

Later, Ramon Lull (d. 1315) imagined machines that could perform reasoning. Thomas Hobbes (1588–1679) compared thinking to math, suggesting that reasoning works like "adding and subtracting ideas." Meanwhile, mechanical calculators were being invented. Leonardo da Vinci (1452–1519) designed one, though he never built it. The first working calculator was made by Wilhelm Schickard (1592–1635), followed by Blaise Pascal (1623–1662) Pascaline in 1642.

The Power of Data in Modern AI:

Early AI relied on hand-coded rules, but today, machine learning lets systems improve by analyzing vast amounts of data. For example:

Language Processing: Instead of teaching a program grammar rules, researchers found that feeding it millions (or billions) of words lets it learn patterns on its own. More data often beats fancy algorithms—even a simple program with tons of data can outperform a sophisticated one with less data.

Image Completion: If you remove an object from a photo, AI can fill the gap convincingly by searching similar images. With just 10,000 photos, results were poor, but with 2 million photos, the system worked amazingly well.

This shift means AI no longer needs every rule programmed manually. Instead, with enough data, machines can learn independently—solving what was once called the "knowledge bottleneck."

Today, AI is everywhere—hidden in industries like healthcare, finance, and technology—making it an invisible but essential part of modern life.

Exercise-01: Is AI a science or is engineering or neither or both ? Explain.

Answer:

Artificial Intelligence (AI) is both a science and an engineering discipline, blending theoretical foundations with practical applications. *As a science*, AI seeks to understand and replicate intelligent behavior through computational models, drawing from mathematics, cognitive psychology, and computer science to explore principles like learning, reasoning, and decision-making. This scientific aspect focuses on questions such as "What is intelligence?" and "How can machines simulate it?"

Simultaneously, *AI is engineering* because it involves designing, building, and deploying systems that solve real-world problems. Engineers apply scientific insights to create practical tools, from chatbots to self-driving cars, emphasizing functionality, scalability, and efficiency. The engineering side asks, "How can we build systems that work reliably in complex environments?"

The interplay between science and engineering in AI is inseparable. Scientific discoveries (e.g., neural networks) drive technological breakthroughs (e.g., deep learning applications), while engineering challenges (e.g., data limitations) inspire new research directions.

Thus, AI transcends the dichotomy—it is a science that advances knowledge and an engineering discipline that transforms theory into tangible innovations.

Exercise-02: Examine the AI literature to discover whether the following tasks can currently be solved by computers.

i) Playing a decent game of table tennis (ping-pong)

ii) Discovering and proving new mathematical theorems

iii) Giving competent legal advice in a specialized area of law

iv) Performing a complex surgical operation.

Answer:

1. Playing a Decent Game of Table Tennis (Ping-Pong):

AI can play table tennis at a competent level, but with limitations. The task involves real-time perception (tracking the ball's speed and spin), rapid decision-making (predicting trajectories), and precise motor control (adjusting paddle angle and force). Modern AI systems, such as robotic arms with vision sensors, can perform well in controlled environments by using reinforcement learning and predictive models. However, they still struggle with unpredictable human strategies, sudden changes in ball dynamics, and long-term adaptability. Unlike chess or Go, table tennis requires continuous, high-speed physical interaction, making it a challenging domain for current AI. While AI can outperform novice players, reaching elite human levels remains difficult due to the need for fine-tuned reflexes and intuitive adjustments.

2. Discovering and Proving New Mathematical Theorems

AI has demonstrated the ability to discover and prove mathematical theorems, but primarily in narrow, well-defined domains. Automated theorem provers (e.g., Lean, Coq) use formal logic to verify proofs, and machine learning models (e.g., DeepMind's work on knot theory) can suggest conjectures by identifying patterns in mathematical data. However, AI lacks the deep conceptual reasoning and creativity of human mathematicians. While it can assist in verifying proofs or exploring combinatorial problems, breakthroughs requiring abstract intuition (e.g., solving the Riemann Hypothesis) are beyond current capabilities. AI's role is more supportive—automating tedious derivations or suggesting hypotheses—rather than independently revolutionizing mathematics.

3. Giving Competent Legal Advice in a Specialized Area

AI can provide legal advice in specialized areas, but with significant caveats. Natural language processing (NLP) models can analyze case law, statutes, and contracts to generate relevant legal arguments or predict case outcomes. Systems like ROSS Intelligence (powered by IBM Watson) assist lawyers in research and drafting. However, AI lacks true legal reasoning—it cannot understand ethics, judge credibility, or adapt to novel legal interpretations. Its advice is based on statistical patterns rather than principled jurisprudence, making it unreliable for high-stakes decisions. While useful for routine tasks (contract review, precedent retrieval), AI cannot replace human lawyers in complex litigation or negotiations where judgment and persuasion are key.

4. Performing a Complex Surgical Operation

AI and robotics can perform complex surgical operations, but typically under human supervision. Robotic systems like the **da Vinci Surgical System** enhance precision in minimally invasive procedures, reducing human error. AI assists in preoperative planning (e.g., tumor segmentation in MRI scans) and intraoperative guidance (real-time tissue recognition). However, fully autonomous surgery remains rare due to safety, ethical, and regulatory concerns. AI lacks the adaptability to handle unforeseen complications (e.g., unexpected bleeding) or make ethical decisions (e.g., prioritizing one surgical goal over another). Current applications focus on

augmenting surgeons rather than replacing them, ensuring human oversight in critical decision-making.

1.2 Intelligent Agents: Agents and environment:

An agent is something that can see or sense what is happening around it and then do something in response. It uses sensors to sense things and actuators to take action. For example, a robot can have cameras as sensors to see and wheels or arms as actuators to move or pick things up. Even a human can be called an agent—our eyes and ears are sensors, and our hands, legs, and voice are actuators that let us do things based on what we see or hear. A software agent, like a program, uses inputs such as keyboard strokes and gives outputs like displaying text or sending data. So, an agent is just anything that can observe its surroundings and take actions based on what it notices.

In artificial intelligence, four important concepts help explain how an agent makes decisions:

i. Percept
ii. Percept Sequence
iii. Agent Function and
iv. Agent Program.

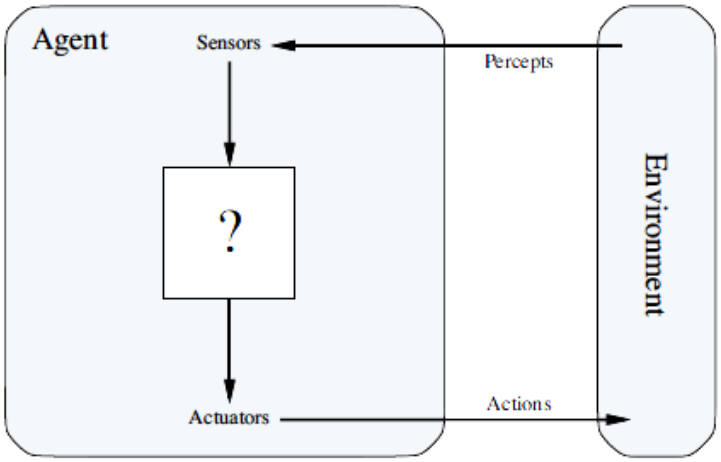
A percept is what the agent senses at a particular moment. For example, if a robot vacuum cleaner is in a room and sees that the floor is dirty, that is its current percept.

A percept sequence is the complete history of everything the agent has sensed so far. So, if the robot first saw a clean room, then a dirty one, then a clean one again, that entire record is its percept sequence. Based on this sequence, the agent needs to decide what to do next.

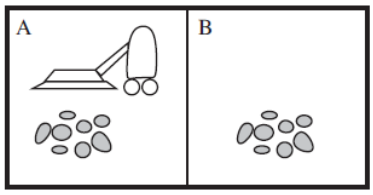
The agent function is like a giant set of rules or a table that says: "If the agent has seen this sequence of events, it should do this action." For instance, if the last two rooms were dirty, the agent function might say “keep cleaning.” However, this table could be huge and difficult to create manually.

That’s where the agent program comes in—it is the actual code or instructions that run inside the agent and decide what action to take, based on the current percept or the percept sequence. So, in our robot vacuum example, the agent program would say: “If the current room is dirty, suck up the dirt; otherwise, move to the next room.” The agent program is the working part that brings the agent function to life inside a machine.

The following figure shows how an agent interacts with its environment through a continuous cycle of sensing and acting.



The environment sends information to the agent through percepts**,** which are picked up by the agent's sensors (like eyes or cameras). Inside the agent (represented by the question mark), it processes these percepts and decides what to do next. The decision leads to actions**,** which are carried out using the agent's actuators (like hands, wheels, or software commands). These actions affect the environment, and the cycle repeats. This loop of sensing and acting is how agents operate and adapt to the world around them.
An example to explain the above topic is shown in the following figure.



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

The above figure shows a simple example used to explain how an intelligent agent, like a robot, can work. It represents a vacuum-cleaner world with only two locations**,** labeled A and B**.** Each location can either be clean or dirty**,** and the job of the vacuum cleaner is to keep the area clean. In the figure, the vacuum cleaner is currently in location A, and both locations have dirt on the floor. The vacuum cleaner can sense where it is and whether that square is dirty. Based on what it senses, it can take actions like cleaning the dirt (suck) or moving to the other location (left or right)**.** This simple setup helps in understanding how an agent makes decisions based on its environment to achieve a goal—in this case, keeping both A and B clean.
It shows a simple rule table for a vacuum-cleaner agent. On the left, we have different percept sequences—what the agent has seen so far (like being in room A and seeing it dirty). On the right, we have the action the agent should take (like "Suck" to clean or "Right" to move). For example, if the agent sees [A, Dirty], it cleans; if it sees [B, Clean], it moves left. This table helps the agent decide what to do based on its past experiences.

1.3 Concept of Rationality:

This section explains what it means for an agent (like a robot or a software program) to behave in a smart and useful way, which we call rational behavior.
A rational agent is one that always tries to do the best possible thing to reach its goal, based on what it knows and what it sees happening around it.
For example, if a robot vacuum cleaner is trying to keep the floor clean, it should suck up dirt when it sees it, avoid wasting time on already clean spots, and stop moving when the job is done. It shouldn't clean the same place again and again just to appear active—that wouldn't be rational.
To know whether an agent is doing well, we use a performance measure. Think of it like a report card or a score system that judges how successful the agent is.

In the vacuum cleaner example, a good performance measure might be "how clean the floor stays over time" rather than "how much it moves or cleans."

Now, a rational agent doesn't need to be perfect or know everything. It only needs to make the best decision with the information it has.

For instance, imagine you're crossing the road and suddenly something unexpected object falls from the sky and hits you. That doesn't make your decision irrational—you simply didn't know it was coming. Similarly, a rational agent makes choices based on what it has seen or experienced so far, not on the future it can't predict.

A smart agent should also be able to learn from its experience. If a vacuum cleaner notices that some areas get dirty more often, it should learn to visit those places more frequently. This ability to improve over time helps the agent become more autonomous, meaning it doesn't rely completely on its maker to tell it what to do in every situation.

In conclusion, a rational agent is one that acts in a way that is most likely to achieve its goals, using the knowledge it has, the tools it's been given, and the things it senses around it. It doesn't have to be perfect, just smart and adaptable in its decisions.

1.4 The Nature of Environment:

1.4.1 Specifying the Task Environment:

Before we can build a smart agent, we need to clearly define the world it will work in and what exactly we want it to do. This is called specifying the task environment, and a useful way to do this is by using the PEAS framework, which stands for Performance measure, Environment, Actuators, and Sensors**.** These four components help us fully understand the agent's job and its working conditions.

The performance measure tells us what counts as success for the agent. For example, in the case of a self-driving taxi, we might want it to complete trips safely, quickly, legally, and with passenger comfort.

The environment includes everything around the agent that it needs to deal with—like roads, traffic, pedestrians, traffic signals, and customers.

Actuators are the parts of the agent that let it take action in the environment. In the self-driving car, these include the steering wheel, accelerator, brakes, and even the turn signals.

Sensors are how the agent collects information from the world—for example, the taxi would use cameras, GPS, speedometers, and microphones to sense what's going on around it.

By carefully specifying these four parts of the task environment, we create a clear picture of what the agent is supposed to do and what tools it has to do it. This makes it much easier to design an intelligent agent that can act appropriately and make smart decisions in its environment.

Exercise: For each of the following activities, give a PEAS description.

i. ***Playing a tennis match***
ii. ***Performing a high jump***
iii. ***Bidding on an item in an auction.***
iv. ***Softbots***

Agent Type	Performance Measure	Environment	Actuators	Sensors
Automated Taxi	Safe, fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers	Steering wheel, accelerator, brake, turn signals, horn, display	Cameras, radar, speedometer, GPS, odometer, accelerometer, engine sensors, microphone
Tennis Player	Win matches, minimize errors, maintain endurance	Court surface, net, opponent, ball, weather	Tennis racket, footwork, body positioning	Eyesight, proprioception, hearing, sweat sensors
High Jumper	Clear maximum height, maintain technique, avoid injuries	Bar height, runway, landing mat, weather	Leg muscles, arm movements, body control	Depth perception, muscle feedback, wind sensors
Auction Bidder	Win desired items, minimize expenditure, detect bluffing	Auction room, other bidders, auctioneer, items	Bid paddle, communication devices	Price displays, facial recognition, voice analysis
Softbots (software agent)	Accuracy of relevant articles, user satisfaction scores	Internet (news sites, blogs, social media)	Display headlines, send notifications, open article links	User clicks, search queries, website data feeds, reading time metrics
Biometric Authentication System	Accuracy of authentication (minimizing false acceptances/rejections), Speed of verification (response time), Security level (resistance to spoofing), User convenience (ease of use).	Physical (e.g., office, smartphone, border control checkpoint), Digital (e.g., login portals, secure databases), Variable lighting/noise conditions (for voice/facial recognition), Dynamic (multiple users, changing biometric data over time).	Lock/unlock mechanisms (doors, devices), Notification systems (alerts for failed attempts), Access control interfaces (grant/deny permissions), Database updaters (store/update biometric templates).	Biometric scanners (fingerprint, iris, facial recognition cameras), Microphones (voice recognition), Touchscreens/pads (palm/vein patterns), Environmental sensors (ambient light/noise adjusters).

1.4.2 Properties of Task Environment:

To design an intelligent agent the different characteristics or properties of task environment should be understood. Some of the most important properties are explained below.

1. Fully Observable vs. Partially Observable

In a fully observable environment, the agent can see everything it needs to make a decision. For example, in a chess game, all the pieces are visible on the board, so the player knows exactly what is going on. But in a partially observable environment, the agent can only see part of the situation. A good example is a self-driving car, which can't see everything around corners or behind other vehicles, so it has to make decisions with limited information.

2. Single-Agent vs. Multi-Agent

In a single-agent environment, the agent works alone and doesn't have to worry about others. For example, a robot solving a maze by itself is a single-agent scenario. In a multi-agent environment, the agent has to consider the actions of others. A good example is a video game or soccer match, where multiple players (agents) are interacting and competing or cooperating with each other.

3. Deterministic vs. Stochastic

A deterministic environment means that actions always lead to the same result. For instance, when you press a button on a microwave, it starts heating—that's predictable. A stochastic environment has unpredictable outcomes. For example, in a weather simulation, even if the conditions are similar, the weather can change unexpectedly, and the agent must be ready to handle different results.

4. Episodic vs. Sequential

In an episodic environment, each action or task is separate from the next one. For example, a spam filter checks each email separately; the decision about one email doesn't affect the next. But in a sequential environment, past actions affect future ones. A good example is playing a video game, where each move influences what happens next.

5. Static vs. Dynamic

A static environment stays the same while the agent is thinking. For example, solving a puzzle on paper—nothing changes unless you make a move. In a dynamic environment, things keep changing on their own. For instance, a robot in a factory must act quickly because machines and people around it are always moving.

6. Discrete vs. Continuous

A discrete environment has clear, separate steps or values. For example, a board game like Ludo or Chess has a limited number of moves and pieces. A continuous environment has many small changes that can happen at any moment. Driving a car is a continuous task because speed, position, and direction can change smoothly at any time.

7. Known vs. Unknown

In a known environment, the agent already understands how things work. For example, a calculator always knows what will happen when you press a button. In an unknown environment, the agent must figure out how the system works. A good example is a robot entering a new room it has never seen before—it doesn't know where things are or what will happen when it moves, so it needs to learn and adapt.

1.5 Structure of AI Agents:

The structure of an agent includes two main parts: the architecture and the agent program. The architecture is the physical system (like sensors, actuators, and hardware), and the agent program is the software that decides what action to take based on sensor input. Together, they make a working agent:

Agent = Architecture + Program.

Agent programs are the software or instructions that tell the agent what action to take based on what it senses (called percepts). While the agent function is like a big lookup table of all possible situations and actions.

The agent program is a practical and efficient way to do this using code that runs inside the agent's system. It helps the agent choose the right actions without needing to store a huge table of possibilities.

There are different types of agent programs, each with its own way of deciding what to do:

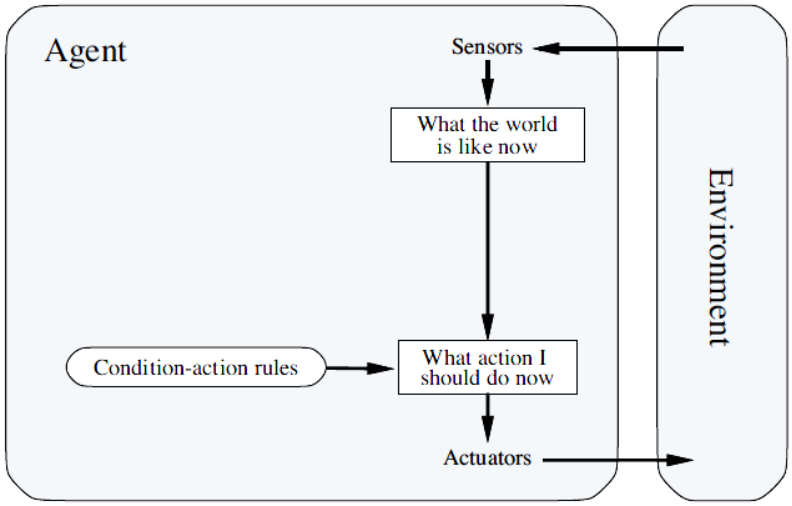
i. Simple Reflex Agents – These respond directly to the current percept using condition–action rules (e.g., if dirty → suck).
ii. Model-Based Reflex Agents – These keep track of the world by storing some information about past percepts, helping them handle partially observable environments.
iii. Goal-Based Agents – These agents act by considering their goals and choosing actions that help them reach those goals.
iv. Utility-Based Agents – These go a step further by not just aiming for goals but also trying to maximize happiness or usefulness (utility) from their actions.
v. Learning Agents – These can improve their performance over time by learning from experience.

Each type is more advanced than the previous one and is used depending on how complex the environment and the agent's task are.

1.5.1 Simple Reflex Agents:

Simple reflex agents are the most basic type of intelligent agents. They choose actions based only on the current situation (called a percept) without considering what happened before. These agents follow condition–action rules, meaning they check what they are sensing and perform an action if it matches a specific condition.

The following figure shows the general structure of a simple reflex agent.



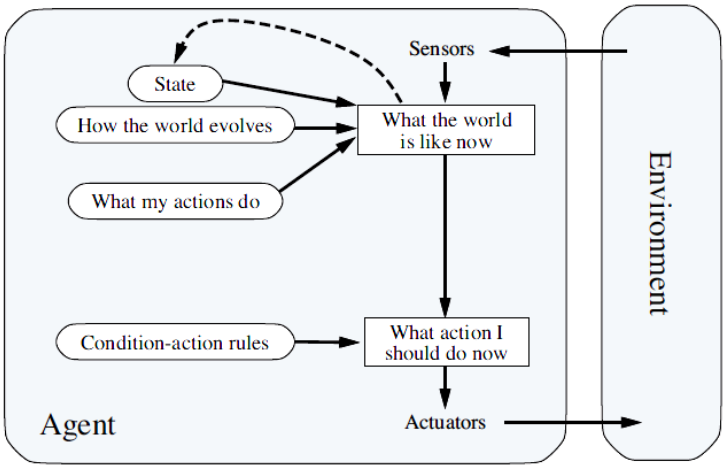
The above figure illustrates the structure of a simple reflex agent, which operates by directly mapping current percepts to actions using predefined condition-action rules. The agent interacts with its environment through sensors, which provide the current percept (e.g., detecting dirt or location in the vacuum-cleaner world).

This percept is interpreted to determine the agent's immediate state (e.g., "location A is dirty"). The agent then matches this state to a set of condition-action rules (e.g., "if dirty, suck; else move"). When a matching rule is found, the corresponding action is executed via actuators (e.g., cleaning or moving).

This design is very fast and easy to use in small, fully observable environments. But it lacks internal state or memory, relying solely on the current percept to trigger reactive behavior. While efficient in fully observable environments, such agents struggle with partial observability, as they cannot account for historical context or unobserved aspects of the environment. For instance, the vacuum agent uses this architecture, but its rationality breaks down if sensor data is incomplete or noisy.

1.5.2 Model-Based Reflex Agents

Model based reflex agent is an extended form of simple reflex agents in which an internal state and a model of the world is incorporated to handle partially observable environments. As illustrated in the following figure.



It can be seen from the figure that these agents interact with their environment through sensors (to perceive inputs) and actuators (to execute actions), but they also maintain an internal representation of the world's state. This internal state is updated dynamically using two types of knowledge encoded in the agent's model:

i. How the world evolves independently (e.g., predicting traffic movement over time).
ii. How the agent's actions affect the world (e.g., turning the steering wheel changes the car's direction).

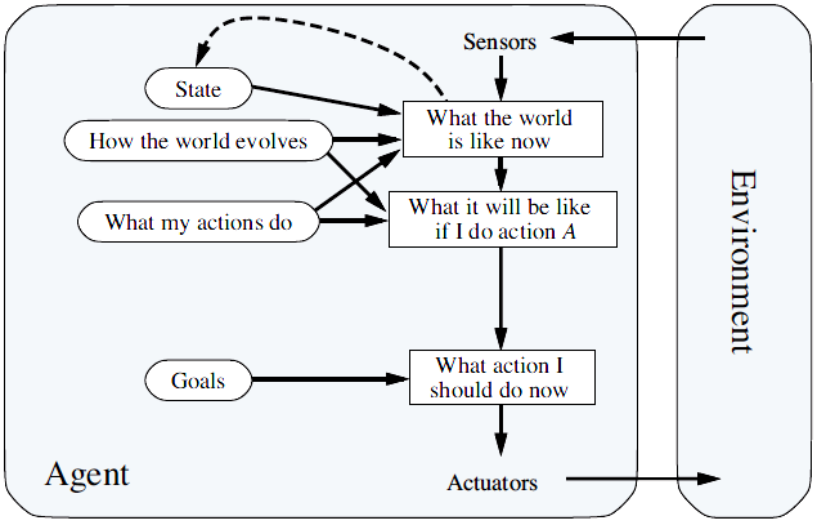
When the agent receives a percept, it combines this new information with its existing internal state to infer a more accurate representation of the current environment.

For example, a self-driving car might use its model to track the positions of vehicles obscured by obstacles, even if they are not directly observed. The updated state is then matched to predefined condition-action rules to select an appropriate action, similar to simple reflex agents.

This approach addresses the limitations of purely reactive agents by enabling the agent to "fill in gaps" in perception using historical data and predictive modeling. By integrating memory and reasoning about environmental dynamics, model-based reflex agents achieve greater adaptability in complex, partially observable settings compared to their simpler counterparts.

1.5.3 Goal Based Agents:

A goal-based agent is a modified version of model-based agent in which the condition action rules are replaced with a specific goal . It uses its understanding of the environment (the model) and plans actions to achieve a specific goal as illustrated in the following figure.



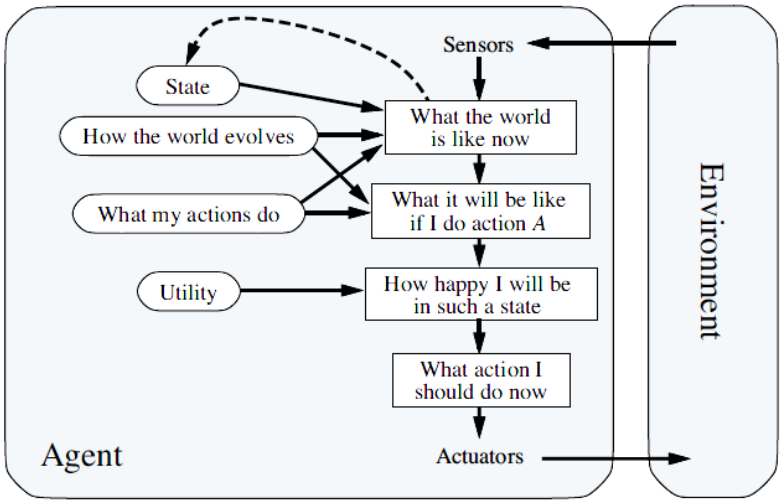
In this figure, the agent first receives input from the environment through percepts (what it senses right now). It then interprets these percepts using an internal model of the world.

Next, the agent checks its goal—what it is trying to achieve. Based on both the current state (from the model) and the goal, the agent searches or plans a sequence of actions that will help it reach the goal. It predicts "If I do this action, what will happen?" and "Will that help me achieve the goal?" before choosing the best action.

For example, think of a driverless car at a crossroads. The car uses its sensors (cameras, GPS) to perceive its surroundings. It updates its model to know where it is, even if it can't see every road. Then, by checking its goal (the passenger's destination), it plans whether it should turn left, right, or go straight to move closer to the goal. A goal-based agent uses a model of the world to plan smart actions that help achieve a goal. Although the goal-based agent appears less efficient, it is more flexible because the knowledge that supports its decisions is represented explicitly and can be modified. On the other hand for the reflex agent, we would have to rewrite many condition–action rules.

1.5.4 Utility Based Agents:

Utility-based agents are more smarter type of agent compared to goal-based agents. A goal-based agent simply tries to reach a goal, but sometimes, there can be many possible goal states, and some might be better than others. This is where a utility-based agent becomes important as illustrated in the following figure.



The above figure shows the structure of a utility-based agent. It looks similar to the goal-based agent, but here, after considering possible future actions, the agent evaluates each possible outcome using the utility function. It then selects the action that leads to the highest utility. The agent uses its model of the environment (like a model-based agent) to predict the results of actions and then compares them based on how desirable those results are according to the utility function.

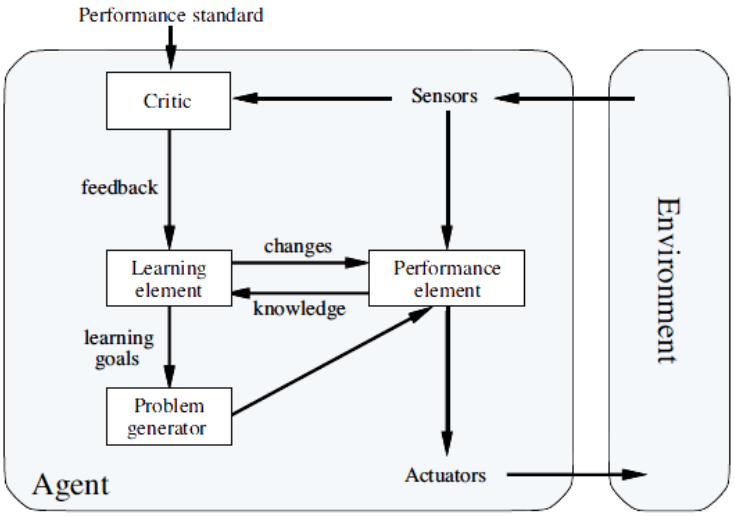
A utility-based agent doesn't just aim to achieve a goal—it aims to achieve the most "satisfying" or "useful" outcome according to a utility function. The utility function measures how good or desirable a certain state is.

For example, in a self-driving car, reaching the destination is the goal. But there can be many ways to do it—some routes may be faster, safer, or more comfortable. A utility-based agent would choose the route that maximizes overall satisfaction (like shortest time, least traffic, safest path), not just any path that reaches the destination.

Thus, while goal-based agents only ask, "Does this action achieve my goal?", utility-based agents ask, "How good is the result if I take this action?" and pick the best among many possible paths. This makes utility-based agents more flexible, smarter, and able to make better decisions, especially in complex environments where multiple outcomes are possible and not all are equally good.

1.5.5 Learning Agents:

Learning agents are those agents which can improve its behavior over time by learning from experience. Unlike basic agents that rely only on fixed rules or models, a learning agent can adapt when it faces new or changing environments as illustrated in the following figure



The above figure shows the structure of a learning agent, which has four main parts:

i. Performance element
ii. Learning element
iii. Critic and
iv. Problem generator.

The performance element is responsible for choosing actions based on the current percept (like a normal agent). The learning element improves the performance element over time by updating how it acts based on feedback. The critic evaluates how well the agent is doing toward its goals and provides feedback. The problem generator suggests new experiences for the agent to explore so that it can discover better strategies.

Taking the driverless taxi example, initially, the taxi might not know the best way to avoid traffic or the smoothest routes. The performance element will drive the car based on its current knowledge (for example, following GPS instructions). The learning element will monitor how well the drives go—if the taxi keeps getting stuck in traffic, the learning element adjusts its driving strategy. The critic constantly checks if the ride was fast, safe, and comfortable. If not, it informs the learning element that the agent needs improvement. The problem generator might suggest that the taxi tries different routes or driving styles (like avoiding main roads during peak hours) to learn new, better ways to complete trips.

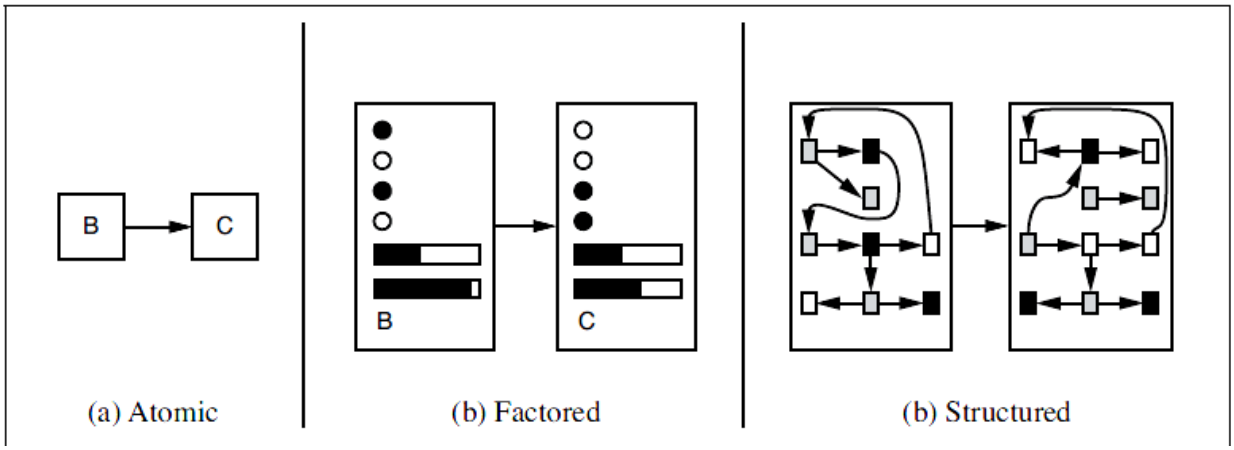
In this way, the above figure shows that a learning agent doesn't stay stuck with its original programming. It keeps improving, adapting to new situations, and becomes smarter over time by learning from its own successes and mistakes—just like how a driverless taxi gets better at handling traffic, weather conditions, and passenger preferences.

1.5.6 Working Principle of AI Agents:

To make an intelligent and capable AI agent, it is necessary to understand the working principle of the various AI components found in an AI Agent. In an AI Agent different components work together to answer important questions like:

- "What is the world like now?"
- "What action should I do now?"
- "What will happen if I do this action?"

The following figure shows three different ways an agent can represent the state of the world and how states change:



i. **Atomic Representation:** In atomic representation, each state is treated like a black box. The agent just knows it is in some state (like "State B" or "State C") but doesn't know anything about what's inside that state. There's no internal details.
Example: A driverless car simply knows, "I am at location X," but nothing about the traffic, fuel, or weather.

ii. **Factored Representation:** In this, the state is broken into attributes or factors. Each factor has a value. These values can be true/false (Boolean), numbers (real-valued), or selected from a list.
Example: The driverless car knows, "I am at location X, speed is 40 km/h, fuel is low, weather is rainy." Each of these is an attribute with its own value.

iii. **Structured Representation:** In structured representation, the state includes different objects (like cars, pedestrians, buildings) and shows relationships between them. Each object can also have its own properties (attributes).
Example: The driverless car knows, "There's a pedestrian crossing ahead, a car next to me, and a red traffic light in front." It understands the detailed layout and connections between objects.
In simple words:

- Atomic = Just knowing which state you're in, nothing more.
- Factored = Knowing the detailed properties of the current situation.
- Structured = Knowing about objects and how they relate to each other.

QUESTION BANK:

Q1: Explain the concept of Artificial Intelligence. Why is it considered one of the most exciting fields in computer science and engineering? Illustrate its goals, scope, and historical background.

Q2: Describe the four main approaches to Artificial Intelligence: Thinking Humanly, Thinking Rationally, Acting Humanly, and Acting Rationally. Give suitable definitions and examples for each

Q3: What is the Turing Test? Explain how it is used to evaluate machine intelligence and discuss its limitations. What are the capabilities a machine must have to pass the Total Turing Test?

Q4: Discuss the "Thinking Humanly" and "Thinking Rationally" approaches to AI. How do cognitive science and logical reasoning contribute to the development of intelligent systems?

Q5: Define a rational agent. How does it differ from other AI approaches? Discuss its components and explain why the rational agent model is more general and scientific.

Q6: Explain the historical and disciplinary foundations of Artificial Intelligence. Describe how fields such as philosophy, mathematics, economics, neuroscience, psychology, and linguistics have influenced AI.

Q7: Describe current real-world applications of Artificial Intelligence across different fields like robotics, game playing, speech recognition, autonomous planning, and spam filtering. Provide examples.

Q8: What is an agent in AI? Explain how agents perceive and act upon the environment through sensors and actuators. Use the vacuum-cleaner example to illustrate your answer.

Q9: Explain the concept of rationality in intelligent agents. What factors determine rational behavior? Discuss performance measures and the importance of learning and autonomy in rational agents.

Q10: What is the PEAS framework in Artificial Intelligence? Explain how a task environment is defined using PEAS with examples like a taxi-driving agent.

Q11: Discuss the various properties of task environments such as observability, agents, determinism, episodic/sequential nature, dynamics, discreteness, and known/unknown settings. Use examples to illustrate each.

Q12: Explain the structure of intelligent agents. Discuss agent functions, agent programs, and architectures. Why is table-driven agent design impractical?

Q13: Describe different types of agent programs: simple reflex agents, model-based reflex agents, goal-based agents, and utility-based agents. How do they differ in behavior and capability?

MODULE-02 PROBLEM SOLVING

Syllabus: Problem-solving: Problem-solving agents, Example problems, Searching for Solutions Uninformed Search Strategies.

2.1 PROBLEM SOLVING AGENTS:

Problem-solving agents are intelligent agents that make decisions by setting goals and finding the best way to achieve them. For example, imagine an agent in Bengaluru that has a confirmed train ticket from Chennai the next morning. The agent's goal is to reach Chennai in time. Instead of worrying about all possible things to do in Bengaluru, the agent focuses only on actions that help it reach Chennai—like checking road or train routes and planning the best travel option. This helps it ignore unrelated activities and simplify decision-making.

To do this, the agent follows a process:

1. **Goal Formulation** – It first decides what goal it wants to achieve based on the situation.
2. **Problem Formulation** – It then defines the problem by identifying the starting point (initial state), possible actions, and the goal state.
3. **Search** – The agent searches for a sequence of actions (a path) that will lead from the starting point to the goal.
4. **Execution** – Once a solution is found, the agent follows the steps to reach the goal.

The environment is assumed to be simple: fully observable, known, discrete, and predictable (deterministic). This makes it easier for the agent to plan a sequence of actions ahead of time.

2.1.1 Well-defined problems and solutions:

To solve any problem in a structured way, it must be well-defined, which means it should include the following five key components.

i. **Initial State** – The starting point or condition from which the agent begins.
ii. **Actions** – The set of possible moves or steps the agent can take.
iii. **Transition Model** – A description of what happens after each action is taken.
iv. **Goal Test** – A check to determine if the agent has reached its desired goal.
v. **Path Cost** – A measure of how costly a sequence of actions is, based on time, effort, or resources.

A problem is considered well-defined when all these parts are clearly stated, allowing the agent to plan a suitable course of action.

For example, if an agent is trying to travel from Bengaluru to Chennai, the initial state is "being in Bengaluru." The actions include options like taking a train, booking a flight, driving a car, or catching a bus. The transition model explains what each action results in — such as a train taking seven hours, or a flight reaching Chennai in one hour. The goal test checks whether the agent has reached Chennai, confirming the success of the journey. The path cost then compares options by their travel time, fare, or convenience — for example, choosing between a ₹600 train ticket that takes seven hours or a ₹2,000 flight that takes one hour. By defining all these parts, the agent can select the most efficient way to reach Chennai.

2.1.2 Formulating problems

Before solving a problem, the agent needs to formulate it properly, which means creating a simplified model of the real-world situation by focusing only on the relevant details. For example, if the goal is to travel from Bengaluru to Chennai, the agent doesn't need to consider

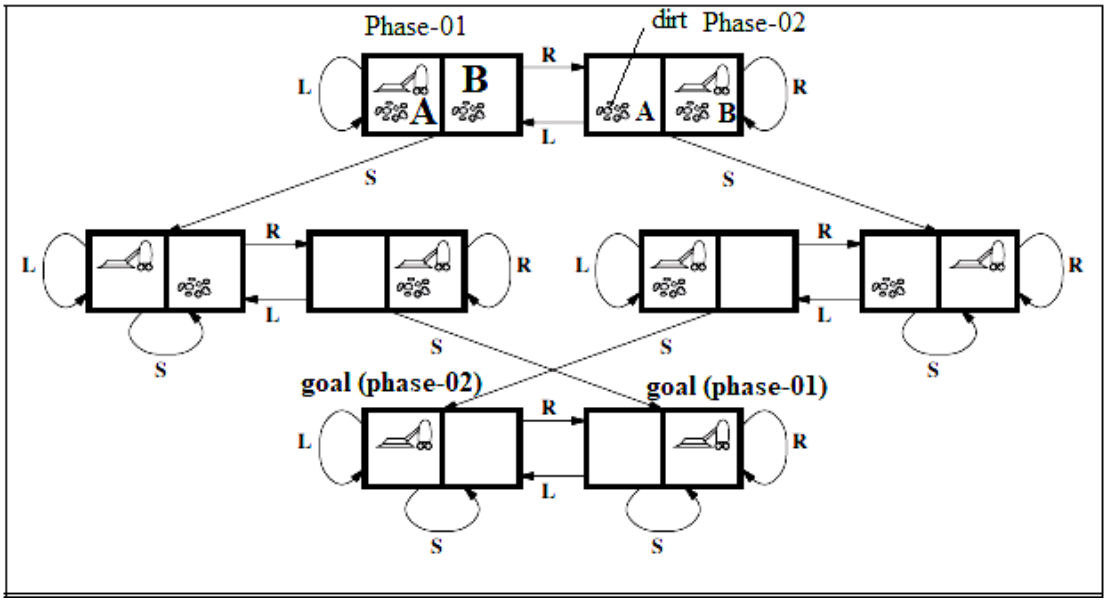
every tiny detail like weather conditions, road signs, or the audio playing in the vehicle. Instead, it creates an abstract model — using cities as states and routes as actions — that helps it plan the trip. It ignores unnecessary factors and only keeps what's needed to find a travel path.

This process of removing irrelevant details is called *abstraction*, and it helps the agent focus on high-level decisions like choosing between taking a train, flight, or bus. As long as the simplified model leads to a workable plan in the real world, the abstraction is valid. This step is crucial because without a clear and useful problem formulation, the agent would be overwhelmed by too much information and unable to act efficiently.

2.2 EXAMPLE PROBLEMS

Example-01: Robot Vacuum Cleaner.

The following figure illustrates a complete, simplified problem model (state space) for the vacuum cleaner agent. It shows how the agent can transition between different situations by performing simple actions, and it is an excellent example of a well-formulated problem using abstraction and action planning.



This diagram shows all the possible states and movements of a simple vacuum cleaner robot in a two-room environment (commonly called Room A and Room B). Each box in the figure represents a state, defined by:

1. The robot's current position (left or right room)
2. The cleanliness status of both rooms (clean or dirty)

There are 8 total states because:

- The robot can be in 2 positions (Left or Right room)
- Each of the 2 rooms can be either clean or dirty (2 options per room) So,

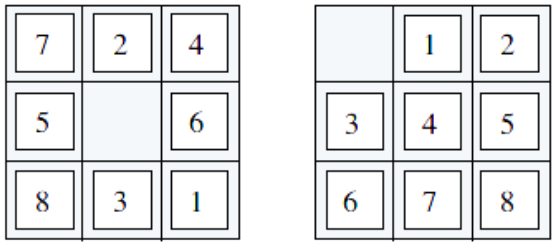
$$2 \text{ (positions)} \times 2 \text{ (A status)} \times 2 \text{ (B status)} = 8 \text{ states}$$

The vacuum icon shows where the robot is. The dots in the room represent dirt. The arrows show possible actions the robot can take L (Left): move to the left room, R (Right): move to the right room and S (Suck): clean the room where the robot is.

Phase-01: In the top-left state, the robot is in the left room (A) and both rooms are dirty. If it uses **S**, the left room becomes clean (middle-left state).Then, if it moves R, it reaches a new state

where the robot is on the right and the left room is already clean. If it now performs **S**, it cleans the right room too — eventually reaching the goal state (phase-01) : both rooms clean. Similarly it applies for the phase-02 position of the robot as well.
The diagram ignores time, battery, or other real-world noise. It focuses only on location and room cleanliness — the minimal information needed to solve the problem. This abstraction helps the agent act faster and more .
All of the five components of a well-defined problem with respect to the above figure is stated below.
i. Initial State – The vacuum starts in a specific room (A or B), with each room either clean or dirty.
ii. Actions – The vacuum can choose to move Left (L), Right (R), or clean the current room using Suck (S).
iii. Transition Model – Each action changes the current state: Suck cleans the room; Left/Right moves the vacuum to the other room without affecting dirt status.
iv. Goal Test – The agent checks whether both Room A and Room B are clean, regardless of its current position.
v. Path Cost – Each action (L, R, S) costs 1 unit, and the total path cost is the number of actions taken to reach the goal.

Example-02: Eight Puzzle.
The 8-puzzle is a sliding tile game on a 3×3 board with 8 numbered tiles and one blank space**.** The goal is to move the tiles around until they are in a specific order (goal state), using only legal moves (sliding a tile into the blank space) as shown in the following fig.



Start State Goal State

i. Initial State
This is how the puzzle starts. In Figure 3.4, the tiles are arranged as shown in the fig.
Here, the blank space (_) is in the middle of the puzzle.
ii. Actions
The agent (player or program) can move the blank space: Up, Down, Left or Right
Only if there's a tile in that direction. Each move slides a tile into the blank spot, effectively swapping them.
iii. Transition Model
Every time the blank moves, it results in a new arrangement of the tiles — a new state. For example:

- If the blank moves right, tile 6 moves left into the blank.
- If the blank moves left, tile 5 moves right into the blank.

This change from one state to another is governed by the transition model.
iv. Goal Test

The agent checks whether the tiles match the goal arrangement shown in the fig.
If the current layout looks exactly like this, the goal is achieved.

v. Path Cost

Each move (sliding a tile) has a cost of 1 unit.The total path cost is the number of moves taken to reach the goal from the start.The aim is to solve the puzzle in the fewest moves possible (least cost path).

Problem: Show that the 8-puzzle states are divided into two disjoint sets, where:Any state is reachable from other states within the same set. No state is reachable from a state in the other set.Also, devise a way to check which set a given state belongs to, and explain why this matters when generating random puzzles.

Solution: In the 8-puzzle, you can only reach a goal state if the number of inversions is even**.** So, the entire state space splits into two disjoint sets**:**

- One with even inversions → Solvable states
- One with odd inversions → Unsolvable states

An inversion is when a higher-numbered tile appears before a lower-numbered one in the list (ignoring the blank). For example in puzzle of 3x3 as shown below

State: 1 2 3
4 5 6
8 7 _

Here, tiles 8 and 7 are inverted (since 8 comes before 7). So, this state has 1 inversion**.**

If a puzzle state has an even number of inversions**,** it belongs to same set (solvable states).

If it has an odd number of inversions**,** it belongs to the other set (unsolvable states).

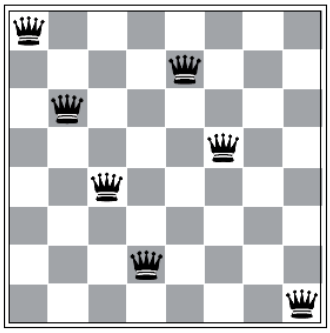
There is no valid sequence of moves that can take a puzzle from one set to the other.

The updated solution with same set (Even inversion) and the other state (odd inversion is shown in the following fig.

Start State	Inversions	Solvability
1 2 3 4 5 6 7 8 _	0	✅ Solvable
1 2 3 4 5 6 8 7 _	1	❌ Unsolvable
1 3 2 4 6 5 7 8 _	2	✅ Solvable

Example-03: Eight Queens Problem:

In the 8-Queens Problem, you start with an empty or partial board, place queens using valid moves, and aim to reach a state where all queens are safe ie none of them can attack each other — no same row, column, or diagonal. The following figure shows a near solution — the task is to fix it using fewer or better-placed actions.



i. Initial State

You start with an empty chessboard, or with a few queens already placed. In the figure, 8 queens are already on the board, but they're not placed correctly.

ii. Actions

You can place a queen in any square on the board — but you must try to avoid putting it where it can attack another queen.

iii. Transition Model

Each time you add or move a queen, the board changes. You go from one arrangement to another, trying to get closer to the correct solution.

iv. Goal Test

You win when all 8 queens are on the board and none of them can attack each other — no same row, column, or diagonal. The board in the figure is close, but still has some queens attacking each other.

v. Path Cost

In this puzzle, path cost is how many steps or moves it takes to place all 8 queens correctly. Fewer moves = better solution.

Example-03: Real World Problems:

Already real world problem in the form of travelling from Bengaluru to Chennai is already explained. It is listed below in terms of those five parameters.

i. Initial State

The agent is currently in Bengaluru and wants to begin the journey.

ii. Actions

The agent can choose from:

- Taking a train
- Booking a flight
- Driving a car
- Catching a bus

iii. Transition Model

This tells what happens after each action:

- Train → Reaches Chennai in 7 hours
- Flight → Reaches Chennai in 1 hour
- Bus or Car → Takes different amounts of time based on traffic

iv. Goal Test

Check whether the agent has reached Chennai.

If yes → Goal achieved.

v. Path Cost

Each option has a different cost:

- Train → ₹600 and 7 hours
- Flight → ₹2,000 and 1 hour
- Car → Fuel cost + time
- Bus → Possibly cheaper but longer

The agent compares these to pick the most efficient option based on time, money, or comfort.

The most optimistic solution is to take a flight from Bengaluru to Chennai**,** which reaches in 1 hour, with a cost of ₹2,000, offering the quickest and most convenient journey.

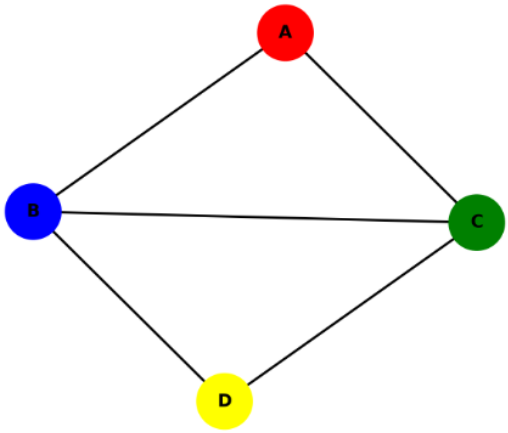
Problem: Give complete problem formulation for each of the following. Choose a formulation that is precise enough to be implemented.

i) Using only four colors, you have to color a planar graph in such a way that no two adjacent regions have the same color.

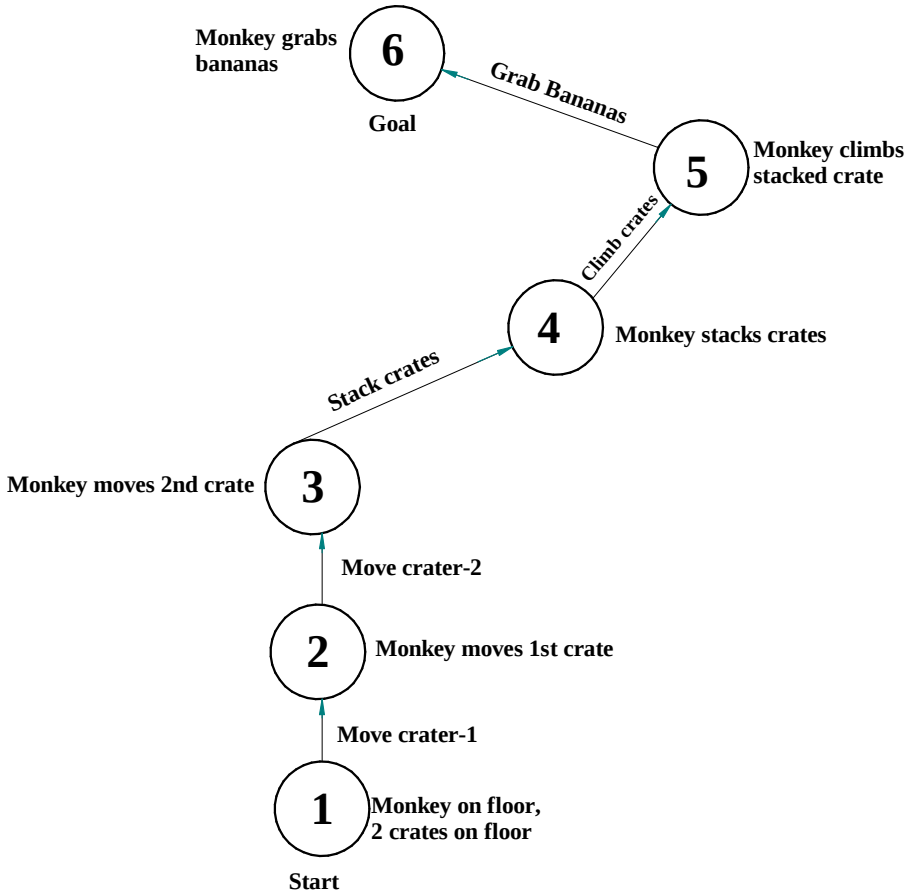
ii) A 3-foot-tall monkey is in a room where some bananas are suspended from the 8-foot ceiling. He would like to get the bananas. The room contains two stackable, moveable, climbable 3-foot high crates.

Solution:

Component	Map Coloring (4-Color Problem)	Monkey and Bananas Problem
Initial State	All regions in the map are uncolored	Monkey is on the floor; 2 crates on the floor; bananas are hanging from ceiling
Actions	Assign one of 4 colors to a region (ensuring adjacent regions differ)	Move crate, stack crate, climb crate, grab bananas
Transition Model	Coloring a region updates the current map state	Each action leads to a new state (e.g., crate moved, crates stacked, etc.)
Goal Test	All regions colored; no two adjacent ones have the same color	Monkey reaches bananas (at 8-foot height) and grabs them
Path Cost	Number of steps (colors assigned) or total constraint checks	Number of actions or energy used (e.g., fewer steps preferred)



Note: If only three colors given, then both A and D can have the same color as it will not break the given rule.



2.3 SEARCHING FOR SOLUTIONS:

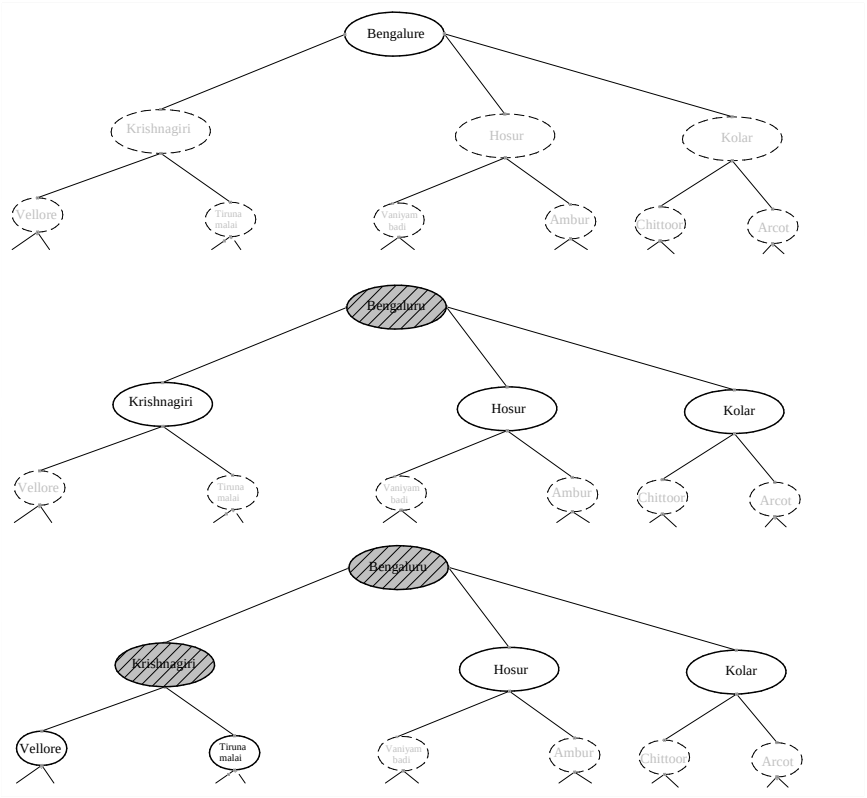
Once a problem is well-defined (with an initial state, actions, goal test, etc.), the next step for an intelligent agent is to search for a solution — that is, a sequence of actions that will lead it from the start state to the goal state. This is done using a structure called a *search tree.*

A **search tree** is a structure used by an agent to explore possible ways to reach a goal. It begins at the **root**, which represents the **initial state** of the problem—for example, starting in Bengaluru. From there, the agent explores **branches**, which represent the different **actions** it can take, like taking a bus, train, or flight. Each resulting point in the tree is a **node**, which represents a new **state**—such as being in Vellore after taking a bus from Bengaluru. As the tree expands, more nodes are added, showing all the paths the agent could take. Some of these are **leaf nodes**, meaning they either reach the goal or have no further actions to take. A complete **path** from the root to a leaf node shows one possible solution to the problem. The search tree helps the agent keep track of what it has tried and find the best way to reach its goal.

To make the search efficient, strategies like *graph search* are used to avoid revisiting the same states. Effective search methods help the agent reach the goal in the shortest or least costly way possible.

2.3.1 EXAMPLE:

The following figure illustrates the concept of search tree for a real time application such as travelling from Bengaluru to Chennai.



a) The Initial State

At the beginning of the search, the agent is located in Bengaluru. This is the root node of the search tree. No moves have been made yet, so the agent's knowledge is limited to its starting location and the fact that it wants to reach Chennai. At this point, the search tree contains only one node — Bengaluru — and no branches.

b) After Expanding Bengaluru

When the agent expands Bengaluru, it considers all the immediate actions available — i.e., the direct cities it can travel to from Bengaluru. These become the first level of child nodes in the search tree. Based on common routes, the agent can choose to go to:

- Krishnagiri (via NH 44)
- Hosur (via expressway)
- Kolar (via NH 75)

Now, the tree has grown — Bengaluru is the parent, and these three cities are its direct children. The agent will now choose one of these to expand further.

c) After Expanding Krishnagiri

Suppose the agent chooses to expand the node Krishnagiri. It now looks at the possible cities that can be reached from Krishnagiri. These become the next set of child nodes:

- Vellore
- Tiruvannamalai

These cities are added below Krishnagiri in the tree. So, now the tree shows:

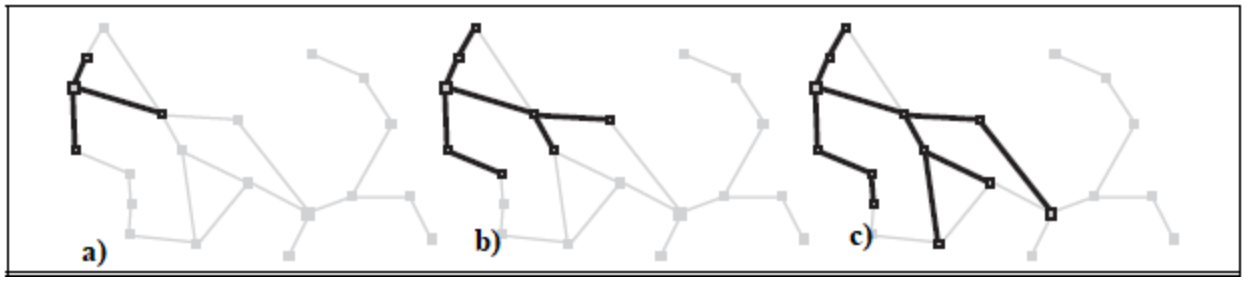
- Bengaluru → Krishnagiri → Vellore
- Bengaluru → Krishnagiri → Tiruvannamalai

These new nodes will later be evaluated to see if they lead to the goal (Chennai).

In the above figure, shaded nodes are states that the agent has already visited and expanded. Outlined nodes are part of the frontier—they've been discovered but not yet explored. Faint dashed nodes are not yet discovered and will only appear when their parent nodes are expanded. These styles show what the agent has explored, what's next, and what's still unknown in the search process.

2.3.2 Graph Search :

Graph search is a search strategy that keeps track of visited states to avoid exploring the same node multiple times. It improves efficiency by eliminating redundant paths and preventing loops. The following figure illustrate graph search strategy for travelling from Bengaluru to Chennai.



a) Initial Expansion from Bengaluru:

In the first tree of Figure 3.8, the agent begins its journey from Bengaluru, which is the starting point or root. At this early stage, the agent chooses one path to expand, say the road to Krishnagiri. So the tree now consists of Bengaluru and one direct connection to Krishnagiri. This represents the very beginning of the graph search, where the agent has made its first move and has just started exploring the map. All other cities are still unknown or unvisited, and only one branch of the tree has been generated.

b) Expanding More Paths:

In the second tree, the agent expands another node — for instance, from Bengaluru to Hosur. Now the tree has two active branches: one from Bengaluru to Krishnagiri, and another from Bengaluru to Hosur. The agent is still exploring immediate neighbors of the starting city. At this point, multiple options are known, and the agent is building up its list of possible paths. It hasn't gone deeply into the map yet, but it has started identifying which cities lie directly on different routes toward Chennai.

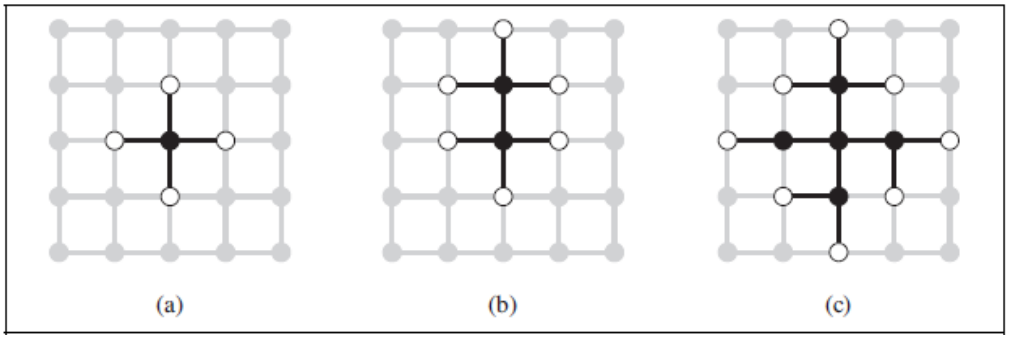
c) Discovering Overlapping Routes (Dead Ends):

By the third tree, the agent starts expanding further. For example, from Krishnagiri it might go to Vellore, and from Hosur to Ambur. Now suppose that another route also leads to Vellore or a nearby city that has already been visited from a different direction. At this point, the agent identifies such a node as a dead end—it realizes that exploring that city again is unnecessary, because its successors have already been explored through another route.

Finally, the agent continues exploring deeper paths. It may now go from Vellore or Ambur to Chennai, the goal city. At this stage, the search graph shows a much clearer picture of the overall route possibilities, but without any duplicated efforts. Cities that were useful are connected, while unnecessary or repeated paths have been skipped. The agent uses this structure to choose the most efficient path—for example, Bengaluru → Krishnagiri → Vellore → Chennai—and reaches the goal successfully.

2.3.3 Separation Property of Graph-Search:

The following figure shows how graph search expands nodes outward from the start, keeping a clear boundary between explored (black), frontier (white), and unexplored (gray) nodes.



In the grid shown, black nodes represent already explored states, white nodes are the frontier (ready to be explored next), and gray nodes are still unexplored. In part (a), only the root node (center) has been expanded, creating a frontier of four adjacent white nodes. In part (b), the agent expands one of those frontier nodes, turning it black and exposing more of the grid. By part (c), all immediate neighbors of the root have been expanded in a clockwise order, pushing the frontier outward. This figure shows how graph search systematically separates the known area from the unknown, expanding the search in a controlled, efficient way without revisiting explored nodes.

The above figure can be compared with a journey from Bengaluru to Chennai. It gradually explores and expand nearby cities. It first expands from Bengaluru to close cities like Hosur and Krishnagiri, then pushes outward to cities like Vellore and Chittoor, always separating explored, frontier, and unexplored locations in an organized way.

2.3.4 Infrastructure for search algorithms

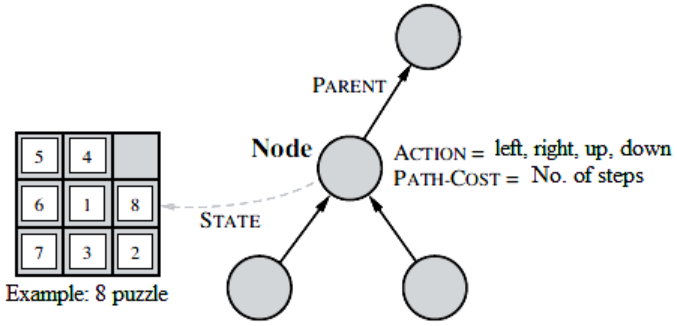
Search algorithms need a special data structure to keep track of the search process as it builds a tree of possible steps. A node holds key information such as

The *current state* (like an 8-puzzle layout),

The *action* that led to it (e.g., moving a tile right),

The *path cost* so far (like the number of steps taken), and

A *link* to the parent node (the previous step) as shown in the following figure.



The arrows in the diagram point from child to parent, which helps the agent, retrace the steps once it finds the goal. Nodes like these form the structure of the search tree**,** allowing the agent to build and track possible paths toward a solution. These components help the agent trace back the solution and calculate the most efficient path to the goal.

In search algorithms, a queue is used to manage which node to explore next. The order in which nodes are removed from the queue determines the search strategy**.** Two common types of queues are FIFO (First-In-First-Out) and LIFO (Last-In-First-Out)**.**

In a FIFO queue**,** the first node added is the first one to be explored. This is like standing in line: whoever comes first is served first. FIFO is used in Breadth-First Search**,** where the agent explores all paths level by level. If multiple nodes are generated from one node (like possible puzzle moves), the agent expands them in the order they were added. In the figure, if multiple nodes are generated from the current node, and you use FIFO, the top node (first added) will be expanded first.

In a LIFO queue**,** the last node added is the first one to be explored, like a stack of plates where the most recent one is taken first. This approach is used in Depth-First Search**,** where the agent goes deep down one path before trying others. If several child nodes are created, the agent starts with the most recently added one, allowing it to dive deeper into a single direction. In the figure, if you generate three child nodes and use LIFO, the last one added (e.g., rightmost) will be the next to expand.

2.3.5 Measuring Problem-Solving Performance

When evaluating how good a search algorithm is, we measure its problem-solving performance using four main factors:

i. Completeness,
ii. Optimality,
iii. Time Complexity, and
iv. Space Complexity.

Completeness means whether the algorithm is guaranteed to find a solution if one exists. For example, if you're planning a trip from Bengaluru to Chennai, a complete algorithm would eventually find a route, even if it’s long.

Optimality checks whether the solution found is the best one, such as the shortest or cheapest route. So, if the algorithm chooses a ₹600 train that takes 7 hours instead of a ₹2000 flight that takes 1 hour, it may not be optimal if your goal is to minimize time.

Time complexity measures how long the algorithm takes to find the solution—important if you need quick answers.

Space complexity refers to how much memory the algorithm uses while searching, especially as it stores routes, cities, and costs. A good search method will balance all these aspects to find the best, fastest, and most efficient path from Bengaluru to Chennai.

2.4 UNINFORMED SEARCH STRATEGIES:

Uninformed search strategies (also called blind search) have no extra information about the goal's location other than the problem definition. They explore the search space blindly.

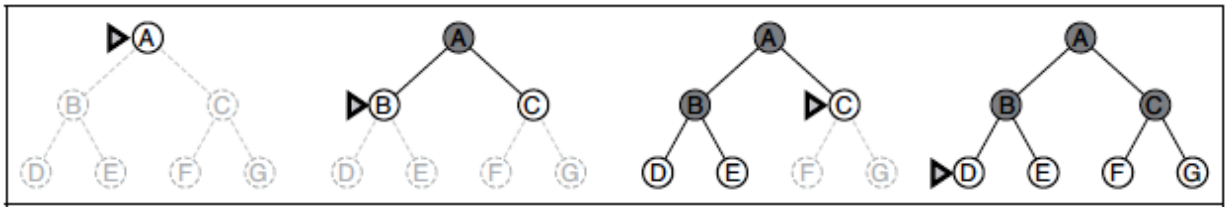
Example: Imagine you are in Bengaluru and want to reach Chennai, but you don’t know the distance, time, or direction. You try each possible route step by step like going to Hosur, Krishnagiri, or Kolar without knowing which one is better. Breadth-First Search or Depth-First Search are examples of this approach.

While on the other hand, Informed search strategies (also called heuristic search) use extra information (heuristics) to estimate how close a state is to the goal, helping the agent make smarter choices.

Example: Now suppose you have a map, and you know the approximate distance or travel time to Chennai from each city. If you’re in Krishnagiri and you see that Vellore is closer to Chennai than Tiruvannamalai, you prefer going through Vellore.

2.4.1 Breadth-First Search (BFS):

Breadth-First Search (BFS) is a basic uninformed search strategy that explores the search tree level by level. It starts from the root node and expands all of its immediate children before moving to the next level as shown in the following fig.

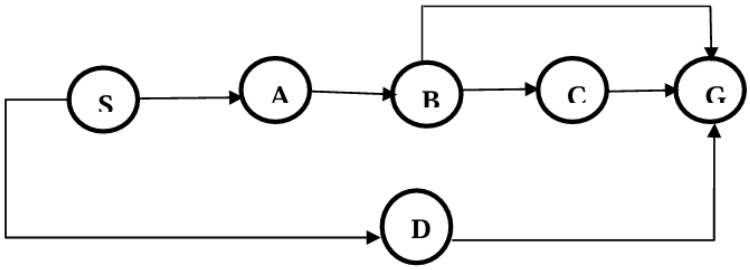


The above figure shows how Breadth-First Search (BFS) works by exploring the tree level by level. The tree starts with node A at the top (the root). In the first stage, A is expanded. Then, in the next stage, its children B and C are added to the queue and B is selected for expansion next. In the following stage, B's children D and E are added, and now C is expanded. After that, C's children F and G are added. Throughout the process, the node marked with the arrowhead indicates which node will be expanded next. This approach uses a FIFO (First-In, First-Out) queue**,** meaning the nodes added first are explored first.

This figure clearly shows BFS's strategy: it expands all nodes at one depth level before moving deeper. So, all children of a node are explored before going to the next level of grandchildren. BFS ensures that the shallowest solution is found first, which makes it complete and optimal (if all steps have equal cost).

However, it can be slow and memory-intensive**,** especially in deep trees, because it stores many nodes at once. For example, if you're planning a trip from Bengaluru to Chennai**,** BFS would try all direct connections first (like Krishnagiri, Hosur), then all cities one step beyond those (like Vellore, Ambur), and continue outward until it reaches Chennai ensuring it finds the shortest route in terms of steps**.**

Problem: For the graph given below apply BFS Search algorithm.



BFS explores level-by-level, from left to right.
It uses a queue (first-in-first-out).
Step 1: Start at S
From S, you have two neighbors:
A
D
So add both A and D to the queue
Now the queue is: [A, D]
Step 2: Visit A
A → B
Add B to the queue
Queue becomes: [D, B]

Step 3: Visit D
D → G
Add G to the queue
Queue becomes: [B, G]
Step 4: Visit B

B → C
Add C to the queue
Queue becomes: [G, C]
Step 5: Visit G
G is the goal node. So stop.
How do we trace the path?
Let's remember how we reached G:
We got to G from D
We got to D from S
So, the full path is: **S → D → G**
This is the shortest and earliest possible route, and BFS always finds this first because it explores all shallow paths first.

2.4.2 Uniform-Cost Search (UCS):

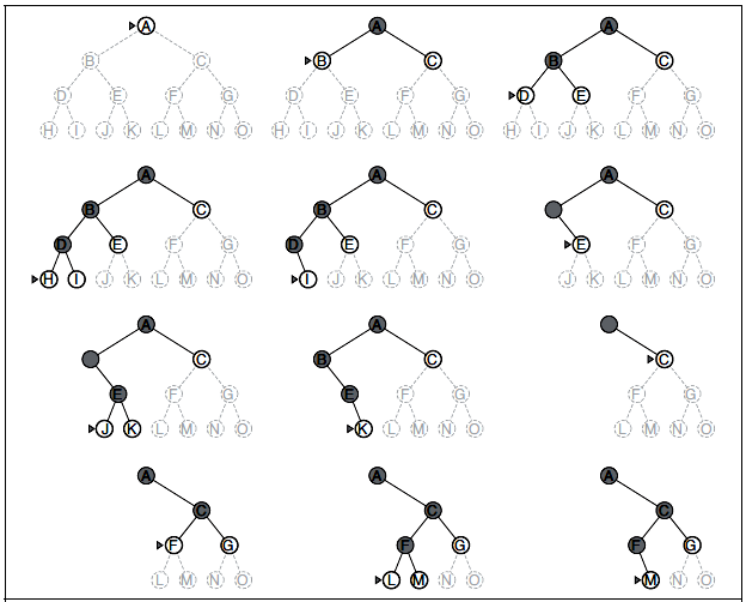
Uniform-Cost Search (UCS) is a search strategy that focuses on finding the least costly path from the start to the goal, regardless of how many steps or levels it takes. Unlike Breadth-First Search, which expands nodes level by level, UCS always chooses to expand the node that has the lowest total path cost so far. It works like someone planning a trip based on total expenses rather than distance or number of stops.

For instance, imagine you're traveling from Bengaluru to Chennai, and you have three possible routes. One route goes through Hosur and Ambur with a total travel cost of ₹900, another goes through Krishnagiri and Vellore costing ₹700, and the third goes through Kolar and Chittoor with a cost of ₹1000. Even if all routes eventually lead to Chennai, UCS will choose the one that gets you there at the lowest cost, which in this case is the ₹700 route via Krishnagiri and Vellore. This approach is especially useful when costs vary — such as fuel expenses, tolls, or ticket prices. UCS guarantees that the first time it reaches the goal (Chennai), it has done so using the cheapest possible path.

Uniform-Cost Search, while guaranteed to find the least-cost path, has several drawbacks. It can be slow because it explores all low-cost paths, even if they don't lead toward the goal, and it often requires high memory to store many possible paths. Since it lacks any guidance toward the goal, it may waste time on irrelevant or long routes**.** Additionally, it depends entirely on having accurate cost information; without it, the search may be inefficient or incorrect.

2.4.3 Depth-first search:

Depth-First Search (DFS) is a search strategy that explores a search tree by going as deep as possible along one path before backtracking. It uses Last-In, First-Out (LIFO) strategy using a stack. This means the most recently generated node is the first to be expanded next. In the following figure , this behavior is clearly illustrated.



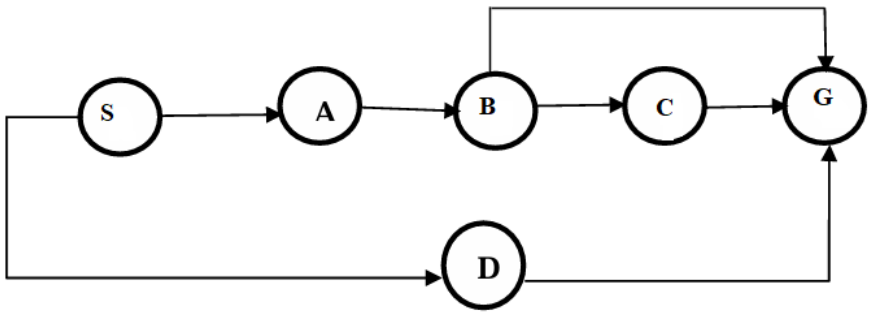
The search starts at the root node A and moves directly down the leftmost path: from A to B, then to D, and finally to H, reaching a depth of 3 levels before backtracking. After reaching a leaf or dead end (no further nodes to explore), DFS backtracks to explore the next sibling node, continuing this process recursively.

Compared to breadth-first search, DFS uses less memory, since it only needs to store nodes along the current path. However, it does not guarantee the shortest or least-cost solution, and if the tree is very deep or infinite, DFS can get stuck exploring one long path forever without finding a solution.

Despite this, DFS is simple and can be useful in cases where memory is limited or the solution is expected to be deep in the tree. DFS is helpful when we only need any solution, not necessarily the shortest or optimal one, and when we want to conserve memory. The above figure helps visualize this by showing how DFS expands nodes deep first, one path at a time, instead of exploring all nodes at a given level like BFS.

In the Bengaluru to Chennai example, DFS might start by going from Bengaluru to Krishnagiri**,** then to Tiruvannamalai**,** and continue down that path without checking other routes like Hosur or Vellore until it hits a dead end.

Problem: For the graph given below apply DFS Search algorithm.



Solution:
It explores as deep as possible before back tracking.It uses a stack (Last-In-First-Out) or recursion.
Step 1: Start at S
From S, let's go to the first neighbor: A
Path so far: S → A

Step 2: Go deeper from A → B
Path so far: S → A → B
Step 3: Go deeper from B → C
Path so far: S → A → B → C
Step 4: Go deeper from C → G
Path so far: **S → A → B → C → G**
Goal node reached!

2.4.4 Depth-Limited Search (DLS):

Depth-Limited Search (DLS) is a variation of Depth-First Search (DFS) that adds a specific limit on how deep the search can go. In DFS, one major issue is that the algorithm can get stuck going down an infinite or very deep path without ever finding the solution.

DLS solves this problem by introducing a depth limit — meaning the agent will only explore paths up to a certain level and then stop. This helps avoid wasting time in very deep or endless branches of the tree.

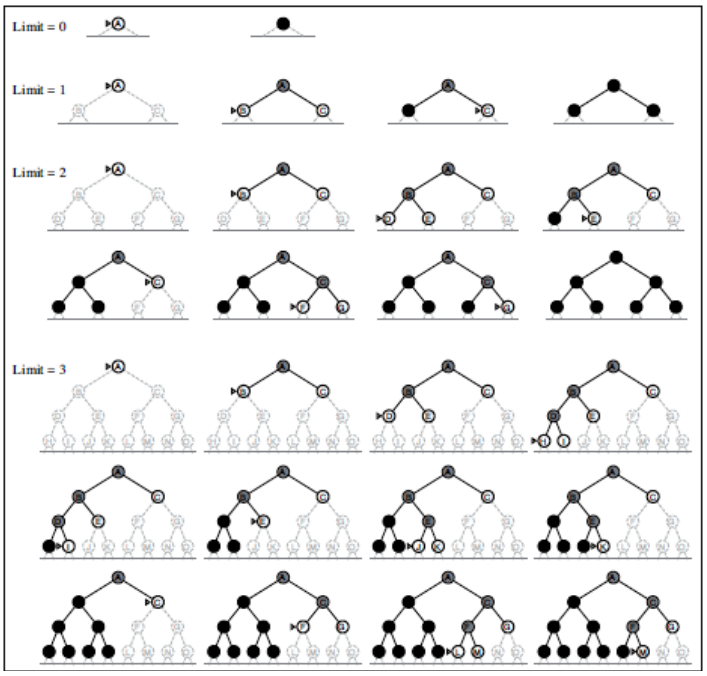
For example, if you're navigating from Bengaluru to Chennai, and you set a depth limit of 3, the search will only explore up to three intermediate cities or levels along any path. If Chennai isn't found within that range, the search stops and either fails or retries with a higher limit.

DLS is especially useful when you have an idea of how deep the solution might be. However, it still shares some drawbacks with DFS, such as not guaranteeing the optimal (shortest or cheapest) path if the solution exists at a shallower level than the limit.

In summary, Depth-Limited Search is a safer and more controlled version of DFS, good for avoiding infinite loops while still conserving memory.

2.4.5 Iterative Deepening Depth-First Search (IDDFS):

Iterative Deepening Depth-First Search (IDDFS) is a search strategy that combines the advantages of Breadth-First Search (completeness and optimality) with the memory efficiency of Depth-First Search. IDDFS works by repeatedly running Depth-Limited Search (DLS) with increasing depth limits. It starts with a depth limit of 0, then 1, then 2, and so on, until the goal is found as shown in the following fig.



Above fig shows a four iteration IDDFS process: The first iteration expands only the root node (depth 0), the second expands all nodes at depth 1, the third includes nodes up to depth 2, and so on. This allows the agent to explore shallow nodes first (like BFS), but without storing all nodes in memory.

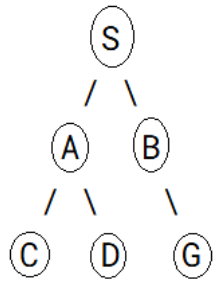
The black dots represent the nodes that are expanded during each. As the depth limit increases step by step, the algorithm explores more levels of the tree. In each iteration, it starts from the root and goes as deep as the current limit allows. The black dots visually show which nodes are visited at that specific depth. For example, in the first iteration (depth 0), only the root node is expanded. In the next iteration (depth 1), the root and its immediate children are expanded. As the limit increases to depth 2, then 3, more nodes are visited further down the tree, and more black dots appear.

For example, if you're finding a route from Bengaluru to Chennai and don't know how many stops are needed, IDDFS tries all routes with 1 stop, then 2 stops, then 3, and continues until Chennai is reached.

This approach ensures that if the goal is at a shallow level, it will be found quickly, and if not, deeper paths will be explored without consuming too much memory. Thus, IDDFS is both complete and optimal (for uniform step costs) and especially effective when the depth of the goal is unknown.

Problem: describe a state space where Iterative Deepening Search (IDS) performs much worse than Depth-First Search (DFS), specifically with time complexities of $O(n^2)$ Vs $O(n)$.

Solution: To explain above we can use a state as shown below with S as starting node and G as goal node.



Start (S): Root of the tree.

Goal (G): Located at depth 2 (right subtree).

Left Subtree: A linear chain (S → A → C or S → A → D) with no goal.

Right Subtree: Short path to the goal (S → B → G).

Behavior of DFS:

DFS might choose the right branch (S → B → G) first and find the goal in 2 steps (O(n), where n is the depth).

If it chooses the left branch first, it will explore S → A → C or S → A → D before backtracking to S → B → G. Still, the time is linear in the depth.

Behavior of IDS:

IDS runs DFS with increasing depth limits:

Depth limit = 0: Explores S. No goal.

Depth limit = 1: Explores S, A, B. No goal.

Depth limit = 2: Explores S, A, B, C, D, G. Finds goal.

At each depth limit, IDS re-explores all nodes up to that limit. For depth *d*, the total work is $1 + 2 + \ldots\ldots\ldots + d = O(d^2)$

If the depth d*d* is proportional to n*n*, the time complexity is $O(n^2)$.

Why IDS is Worse Here:

The goal is at depth 2, but IDS must explore all depths up to 2, redundantly visiting S, A, and B multiple times.

DFS, if it chooses the right branch early, finds the goal in O(n) time without redundant exploration.

A state space where iterative deepening search (IDS) performs much worse than depth-first search (DFS) can be constructed as follows:

The state space is a tree where the goal is located at depth n*n* along a single branch (e.g., the rightmost branch).

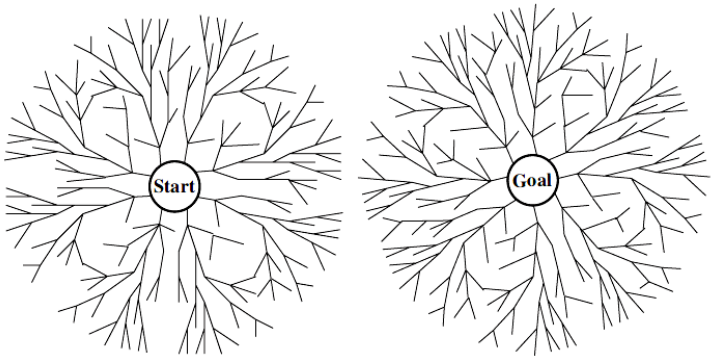
The remaining branches form long linear chains (e.g., left subtrees) that do not lead to the goal.

IDS will explore all depths from 1 to *n*, redundantly visiting nodes at each depth, resulting in $O(n^2)$ time.

DFS, if it chooses the correct branch early, will find the goal in O(n) time without redundant exploration.

2.4.6 Bidirectional Search (BS):

Bidirectional Search is a powerful search strategy that reduces the amount of work needed to find a solution by searching forward from the start state and backward from the goal state at the same time. Instead of exploring the entire path from the beginning to the goal in one direction, this method performs two simultaneous searches: one that begins at the start and moves forward, and another that begins at the goal and moves backward. The search stops when the two meet somewhere in the middle. This drastically reduces the number of nodes explored because each search only needs to go halfway as illustrated in the following figure.



The above figure illustrates this idea clearly: the left tree grows outward from the initial state, while the right tree grows backward from the goal state. Both trees meet in the middle, where their paths connect.

For example, if you're traveling from Bengaluru to Chennai, a bidirectional search would start one search forward from Bengaluru and another backward from Chennai. If both searches reach Vellore, the complete path can be reconstructed from Bengaluru to Chennai by combining the two halves.

This method is highly efficient for problems where the start and goal are known in advance, but it requires the ability to search backward from the goal, which may not always be possible in all problems.

2.4.7 Comparison of Uninformed Search Strategies:

Strategy	Completeness	Optimality	Time Required	Memory Needed	Key Idea
Breadth-First Search	Yes	Yes (if all costs equal)	Very high, especially for deeper problems	Very high, as all nodes at each level are stored	Explores all paths level by level from the start
Uniform-Cost Search	Yes	Yes	High, especially when many cheap paths exist	High, because it stores many paths and costs	Expands the path with the lowest total cost so far
Depth-First Search	No	No	Can be low, but may waste time in deep paths	Low, only stores the current path	Goes deep into one path before backtracking
Depth-Limited Search	No (if limit too small)	No	Limited, depends on depth limit	Low to moderate	DFS with a fixed maximum depth
Iterative Deepening DFS	Yes	Yes (if all costs equal)	Moderate, some repeated effort but efficient	Very low, only current path is stored	Repeats DFS with increasing depth until goal is found
Bidirectional Search	Yes	Yes	Much faster (searches from both ends)	Moderate, keeps track of two frontiers	Searches forward from start and backward from goal

2.5 QUESTION BANK:

Q1: Explain the structure and working of a problem-solving agent. Describe the steps it follows from goal formulation to execution with an example of travelling from Bengaluru to Chennai.

Q2: Define a well-defined problem. Describe each of its five components—initial state, actions, transition model, goal test, and path cost—with a suitable travel example.

Q3: What is problem formulation in artificial intelligence? Explain the importance of abstraction in simplifying problems, using the travel scenario from Bengaluru to Chennai.
Q4: Describe the state space, actions, and goal for the vacuum cleaner problem and the 8-puzzle problem. Discuss how these are examples of well-defined problems.
Q5: What is a search tree? Explain its components including root, nodes, branches, and goal states with the help of the Bengaluru to Chennai travel problem.
Q6: Differentiate between tree search and graph search strategies. Explain the separation property of graph search with a grid or city-based example.
Q7: Explain the internal structure of a node in search algorithms. Discuss the role of queues (FIFO and LIFO) in determining the search strategy with examples.
Q8: Describe the four factors used to evaluate search algorithms—completeness, optimality, time complexity, and space complexity—with reference to the Bengaluru to Chennai travel case.
Q9: Compare uninformed search strategies like BFS, DFS, UCS, DLS, IDDFS, and Bidirectional Search in terms of completeness, optimality, time, space, and their key idea.
Q10: Explain any three uninformed search strategies in detail—such as Breadth-First Search, Uniform-Cost Search, and Depth-First Search—with advantages, limitations, and suitable examples.

MODULE-03: PROBLEM SOLVING & LOGICAL AGENTS

Syllabus: Informed Search Strategies, Heuristic functions. ***Logical Agents:*** *Knowledge −based agents, The Wumpus world, Logic, Propositional logic, Reasoning patterns in Propositional Logic*

3.1 Informed Search Strategies (Heuristic Functions):

Informed or heuristic search strategies use additional knowledge about the problem (beyond just the definition) to find solutions more efficiently than uninformed methods. These strategies rely on a heuristic function, usually written as h(n), which estimates the cost from the current node to the goal.

A heuristic function, written as h(n)**,** is an estimate that helps an agent decide which path to explore first while solving a problem. It gives an idea of *how close a particular state is to the goal***,** based on some extra knowledge.

For example, if you're traveling from Bengaluru to Chennai and you're currently in Vellore, the heuristic could be the straight-line distance from Vellore to Chennai.

Though it may not be exact, a good heuristic guides the search algorithm to make faster and smarter decisions by focusing on promising paths.

One common method is Greedy Best-First Search, which chooses the path that seems closest to the goal based on the heuristic, but it may not always find the best path.

A more powerful strategy is *A* Search (A star search)*, which combines the cost to reach a node (g(n)) and the estimated cost to reach the goal (h(n)), using the formula

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

In A* search, the function f(n) helps decide which path to explore by combining two values: g(n), the actual cost from the start node to the current node, and h(n), the estimated cost from the current node to the goal based on a heuristic. This gives f(n) = g(n) + h(n), the total estimated cost of reaching the goal through that node.

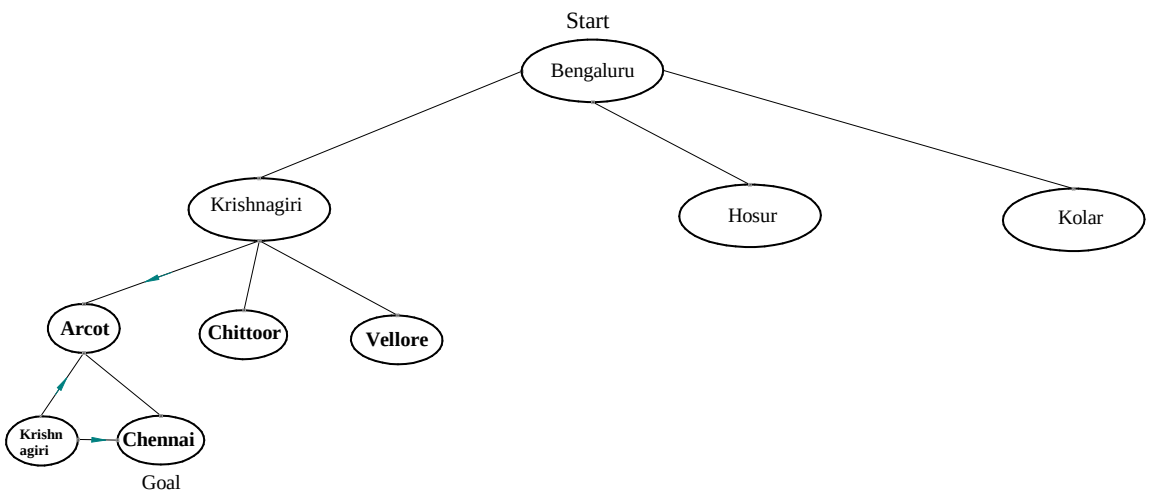
For example, if you're traveling from Bengaluru to Chennai and you're currently in Vellore, g(n) could be the travel cost to reach Vellore, h(n) could be the estimated time or distance remaining to Chennai, and f(n) gives the overall expected cost of that route.

A* chooses the path with the lowest f(n), making the search both smart and efficient when the heuristic is accurate.

However, A*search uses a lot of memory. To handle that, we use memory-bounded algorithms like Recursive Best-First Search (RBFS) and Simplified Memory-Bounded (SMA*), which try to save memory while still searching smartly. Overall, informed search is faster and more goal-focused than uninformed methods, especially when a good heuristic is available. The above concepts are covered in detail in the following sections.

3.2 Greedy Best-First Search

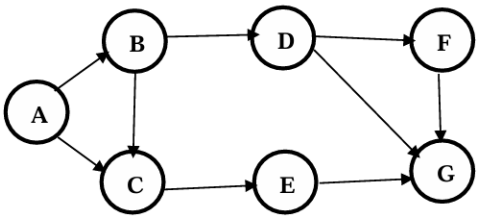
Greedy Best-First Search is an informed search strategy that focuses only on reaching the goal as quickly as possible, based on a heuristic estimate. It selects the next node to expand based on the value of h(n), which is the estimated cost from the current city to the destination as illustrated in the following figure with Bengaluru to Chennai Example.



The above fig. illustrates the working of Greedy Best-First Search**,** where the search algorithm always selects the next city that appears closest to the goal**,** based only on a heuristic estimate such as straight-line distance. There are three possible routes to connect Chennai such as Krishnagiri, Hosur and Kolar. It may choose Krishnagiri as the next step, assuming it's the city with the lowest heuristic value (i.e., closest to Chennai). From Krishnagiri**,** the algorithm may then consider cities like Arcot, Chittoor or Vellore. If Arcot has the smallest estimated distance to Chennai**,** it is selected next. Finally, the path continues from Arcot and then to Chennai**.**

The word "Greedy" is used because the algorithm always picks what seems best at the current step, without thinking about the overall journey.
In summary, Greedy Best-First Search is fast and simple, but can be misleading if the heuristic isn't accurate, as it only follows what looks promising without considering how much effort it actually takes to get there.

Numerical-01: In the below graph, find the path from A to G using greedy best first search algorithm. The values in the table represent heuristic values of reaching the goal node G pass current node.



A	5
B	6
C	4
D	3
E	3
F	1
G	0

Solution:
Start at A → h = 5
Neighbors of A:
B → h = 6
C → h = 4 □ (smaller than B, so selected)
So GBFS picks C and does not expand B, because it simply chooses the most promising-looking node based on lowest h(n).
At C → h = 4
Neighbors: E(h=3)
□ ChooseE
At E → h = 3
Neighbors: G(h=0)
□ ChooseG (goal)

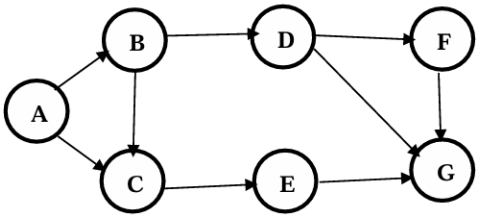
Final Path (GBFS):
A → C → E → G
Total actual cost (g): 1 + 1 + 1 = 3

Note: B and D were not expanded because Greedy Best-First Search only followed the path with the lowest heuristic values.The algorithm found the goal by following A → C → E → G. There was no need to expand other options like B or D because the goal was reached earlier.
GBFS never goes back to explore B or D, because it is a greedy strategy — it always follows the shortest-looking route to the goal based on the heuristic.

3.3 A*Search:

A* search is an informed search algorithm that finds the best path to a goal by combining the actual cost to reach a point and the estimated cost to the goal. It uses the formula:
f(n) = g(n) + h(n) where g(n) is the cost from the start to the current node and h(n) is the estimated cost from the current node to the goal.
It considers both the path already taken and what's ahead. In the above fig in which Greedy Best-First Search was explored, where the next city was chosen purely based on how close it looks to the goal (using the heuristic function h(n)). However, if we apply A* Search to the same tree, the decision at each step would be based on the combined value of the actual cost to reach a city so far (g(n)) and the estimated distance to the goal (h(n)). This means that instead of just picking the city closest to Chennai on the map, A* picks the city with the lowest total estimated cost from start to goal.
For example, starting from Bengaluru, suppose we can go to Krishnagiri, Kolar, or Hosur. Even if Krishnagiri looks closest to Chennai on the map (has the smallest h(n)), it might have a high travel cost to reach from Bengaluru (high g(n)). A would compute f(n) = g(n) + h(n) for each option and might choose Hosur instead if its combined travel cost and distance to goal is lower. As we progress to cities like Vellore, Arcot, and finally Chennai, A* continues to evaluate paths not just based on what looks closer, but what is truly cheaper overall. This leads to better decisions and ensures that the final path from Bengaluru to Chennai is both complete and optimal, unlike Greedy Search which may take a wrong turn based on appearance alone.*
A* search is that it finds the shortest or least-cost path (i.e. it is optimal), while Greedy Best-First Search may not. Greedy search only looks at how close the current city appears to the goal (using the heuristic h(n)), so it can be misled by routes that seem promising but are actually longer or costlier.

Numerical-01: In the below graph, find the path from A to G using A* search algorithm. The values in the table represent heuristic values of reaching the goal node G pass current node.

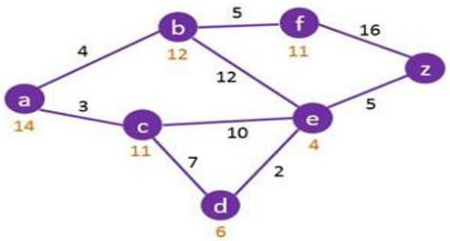


A	5
B	6
C	4
D	3
E	3
F	1
G	0

Solution:
A* uses the formula: f(n) = g(n) + h(n) Where: g(n) = actual cost from start to node n, h(n) = estimated cost from n to goal, A* always picks the node with the lowest f(n) from the OPEN list
From Node A (Start):
g(A) = 0
Neighbors:
B: g=1, h=6 → f = 1 + 6 = 7
C: g=1, h=4 → f = 1 + 4 = 5
 So, A* selectsC because f=5 < f=7
From Node C:
g(C) = 1

Neighbors:
E: g=1 + 1 = 2, h=3 → f = 2 + 3 = 5
 Now we compare E (f=5) and B (f=7) → E is better, so expand E
From Node E:
g(E) = 2
Neighbor: G
g(G) = 2 + 1 = 3, h=0 → f = 3
 G has f=3 → lowest in the open list → expand and goal is reached!
Note: *What About B and D?*
B was added to OPEN list, but its f = 7, which was always greater than f(C) = 5 and f(E) = 5. So, A never had to choose B, because a better (lower f) path was always available. D was never discovered at all — because it is only reachable through B → D, and B was never expanded*

Numerical:-02 Apply the A* search to find the solution path from a to z. Heuristics are with nodes, and cost is with edges. Write all steps as well as open and closed lists for full marks.



Solution:

A* chooses paths using: f(n) = g(n) + h(n)
where: g(n) = actual cost to reach node n from start and h(n) = estimated (heuristic) cost from node n to goal z
Given (from the graph you uploaded):
Heuristic values (h(n)):
h(a) = 14
h(b) = 12
h(c) = 11
h(d) = 6
h(e) = 4
h(f) = 11
h(z) = 0
Edge costs (g):
a → b = 4
a → c = 3
c → d = 7
c → e = 10
b → f = 5
e → z = 5
Step-by-Step A* Path Search
We start at node a.
Step 1: From a
g(a) = 0
Now expand a:
To b: g = 4, h = 12 → f = 4 + 12 = 16
To c: g = 3, h = 11 → f = 3 + 11 = 14
So, from a, we have two options:
b (f=16)
c (f=14) □ Best
Pick c (lowest f)

Step 2: From c
g(c) = 3 (already calculated)
Expand c:
To d: g = 3 + 7 = 10, h = 6 → f = 10 + 6 = 16
To e: g = 3 + 10 = 13, h = 4 → f = 13 + 4 = 17
Now OPEN list has:
b (f=16)
d (f=16)
e (f=17)
Pick b or d (both have f=16)
Let's say we pick d
Step 3: From d
d doesn't lead to goal, and doesn't offer better path
Now back to OPEN list:
b (f=16), e (f=17)
Pick b, but b leads to f →
g(f) = 4 (b) + 5 = 9
h(f) = 11 → f = 9 + 11 = 20
f=20 is worse than other paths.
Now pick e (f=17)
Step 4: From e
g(e) = 13
e → z = 5 → g(z) = 13 + 5 = 18, h(z) = 0
→ f(z) = 18
So we reach goal z!
Final Path: a → c → e → z
a → c = 3
c → e = 10
e → z = 5
Total cost = 3 + 10 + 5 = 18

3.4 Conditions for Optimality while Using A*Search:

For A* search to always find the optimal (least-cost) path, the heuristic function it uses must satisfy two important conditions:

1. Admissibility and
2. Consistency.

A heuristic is said to be admissible if it never overestimates the actual cost to reach the goal from any given node. This means that the estimate provided by the heuristic should always be equal to or less than the true cost.
For example, if you're traveling from Vellore to Chennai, and the actual travel cost is ₹400, the heuristic value h(Vellore) should be ₹400 or less—not more.

The second condition, consistency (also called monotonicity), ensures that the estimated cost from the current node to the goal is no greater than the cost of reaching a neighboring node plus the estimate from that neighbor to the goal. Mathematically, this is written as:

$$h(n) \leq c(n,n') + h(n')$$

Where, h(n) = Estimated cost from the current city to goal, c(n, n′) = Actual cost from the current city to its neighboring city and h(n′) = Estimated cost from that neighboring city to the goal.

Example: **Devise a state space in which A* using GRAPH-SEARCH returns a suboptimal solution with h(n) function that is admissible but inconsistent.**

Let's say: n = Krishnagiri, n′ = Vellore, if

h (Krishnagiri) = Estimated cost to Chennai = ₹300

c (Krishnagiri, Vellore) = Actual cost from Krishnagiri to Vellore = ₹150

h (Vellore) = Estimated cost from Vellore to Chennai = ₹120

Now apply the condition:

$$h(n) \leq c(n, n\prime) + h(n\prime)$$

$$\text{h(Krishnagiri)} \leq \text{c(Krishnagiri, Vellore)} + \text{h(Vellore)}$$

$$300 \leq 150 + 120 = 270 \times$$

This violates the consistency condition, because the heuristic at Krishnagiri is overestimating the cost to the goal compared to the cost of going through Vellore. This could mislead A* into making poor decisions.

Now suppose, Let's say: n = Krishnagiri, n′ = Arcot

h (Krishnagiri) = Estimated cost to Chennai = ₹300

c (Krishnagiri, Arcot) = Actual cost from Krishnagiri to Arcot = ₹150

h (Arcot) = Estimated cost from Arcot to Chennai = ₹170

Now apply the condition:'

$$\text{h(Krishnagiri)} \leq \text{c(Krishnagiri, Arcot)} + \text{h(Arcot)}$$

$$300 \leq 170 + 150 = 320 \checkmark$$

The inequality holds true, which means the heuristic is consistent. This tells A* search that choosing the path from Krishnagiri → Arcot → Chennai is better than taking the route via vellore.

3.5 Memory-Bounded Heuristic Search :

Sometimes, search algorithms like A* need too much memory. To overcome this, we use memory-bounded search methods listed below that work well even with limited memory.

- RBFS (Recursive Best-First Search): It uses less memory by going deep into one path at a time. If that path doesn't work, it comes back and tries another, remembering only the best options. It works like A*, but with less space.
- SMA* (Simplified Memory-Bounded A*): It uses as much memory as allowed. If memory gets full, it removes the least useful paths (those with higher cost) but can bring them back later if needed.

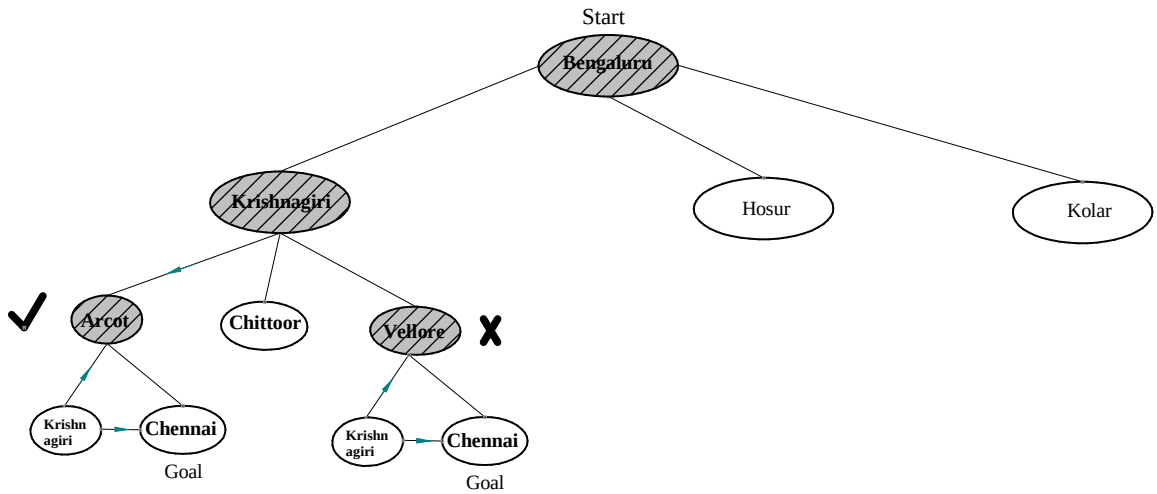
But the problem with RBFS (Recursive Best-First Search) is that it forgets old paths when it goes back, so it may have to repeat the same work again and again, which takes more time while, SMA* is better because it remembers more paths using all the memory it has. If memory is full, it only removes the worst options and keeps the best ones, so it doesn't have to repeat as much work.

Both RBFS and SMA* aim to balance optimal search with limited memory usage, making them suitable for large or complex problems.

3.6 Recursive Best-First Search (RBFS):

Recursive Best-First Search (RBFS) is a memory-efficient version of A* search that explores paths in a depth-first manner, but still uses a heuristic to make smart choices. Unlike A*, which keeps all generated nodes in memory, RBFS only remembers the current path and the best alternative path at each step. The algorithm starts from a root node and explores the child with the lowest estimated cost (f-value). If the best path becomes worse than the second-best option, it backtracks and tries the next best route.

Example: Imagine you are planning a trip from Bengaluru to Chennai and you can go through cities like Krishnagiri, Vellore, and Arcot as shown in the following fig.



RBFS first picks the city that looks most promising based on travel cost plus estimated distance to Chennai. Suppose it chooses the Bengaluru → Krishnagiri → Vellore path. If the cost suddenly becomes too high when reaching Vellore, and the path through Arcot looks better, the algorithm backtracks and explores that alternative and it keeps continuing until it finds the best path.

It keeps only the current path and the best next option in memory, making it memory-efficient. However, the trade-off is that RBFS may have to repeat work, as it discards less promising paths and recalculates them if needed.

3.7 Simplified Memory-Bounded A* (SMA*):

SMA*, or Simplified Memory-Bounded A*, is a smart search algorithm that works like A* but is designed to run well even when memory is limited. It tries to find the best path to a goal while using only as much memory as is available. Just like A*, it uses the formula f(n) = g(n) + h(n), which combines the actual cost so far (g(n)) and the estimated cost to the goal (h(n)).

However, if memory gets full while exploring too many routes, SMA* forgets (removes) the least promising paths—those with the highest f(n) values—and keeps the best ones. If needed later, SMA* can re-expand those forgotten paths by recalling a summary of their cost.

Example: If you are planning a trip from Bengaluru to Chennai and considering different routes—through Krishnagiri, Kolar, or Hosur—SMA* will explore the best-looking routes first. If your system has limited memory and cannot hold all possible paths, SMA* will keep only the most promising cities (say, Vellore → Arcot → Chennai) and remove less useful ones (like Chittoor or Ambur) from memory. If the best path gets blocked or turns out too costly, it can revisit the forgotten ones, if needed. This way, SMA* balances efficiency and memory usage, helping you reach Chennai through the most cost-effective route your system can handle without crashing due to memory overload.

3.8 HEURISTIC FUNCTIONS

In artificial intelligence, especially in informed search strategies like A*, heuristic functions play a crucial role. A heuristic function is a method that gives an estimated cost from the current state to the goal state. It doesn't tell the exact cost, but it gives a smart guess that helps the search algorithm decide which path or move to explore first. A good heuristic helps in reducing the number of steps taken to reach the goal by avoiding unnecessary or unhelpful paths. There are two commonly used heuristic functions such as

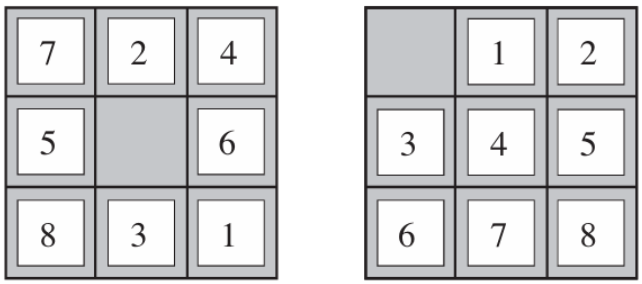
Misplaced Elements Heuristic: h1(n)

$h_1(n)$ is a simple guess of how far a state is from the goal. It just counts how many things are wrong or not in the right place. It doesn't say how far off they are, just that they are not correct. It is quick and easy to calculate, but not very detailed.

Manhattan Distance Heuristic: h2(n)

$h_2(n)$ is a smarter guess that looks at how much work is needed to reach the goal. It adds up the total steps or cost required to fix everything. This makes it more accurate than $h_1(n)$, but it can take a little more time to calculate.

***Example-01:* 8 puzzle problem.**



Start State Goal State

In the 8-puzzle problem as shown above, $h_1(n)$ and $h_2(n)$ are heuristic functions used to estimate how far a given puzzle state is from the goal. $h_1(n)$, known as the misplaced tiles heuristic, simply counts the number of tiles that are not in their correct positions. It's easy to compute but provides only a rough idea of progress.

Out of the 8 tiles (excluding the blank), all 8 tiles are in the wrong position.

Therefore $h_1(n) = 8$

On the other hand, $h_2(n)$, called the Manhattan distance heuristic, calculates the total number of moves each tile needs to reach its goal position, adding both horizontal and vertical steps. While $h_1(n)$ tells how many tiles are wrong, $h_2(n)$ tells how far off they are, making $h_2(n)$ more accurate and effective in guiding search algorithms like A*.

The Manhattan Distance is the total number of moves (horizontal + vertical) each tile must take from its current position to its goal position.
For example Tile 6 is in position: (2,3) → row 2, column 3 when compared to the start state.
While Tile 6 should be in position: (3,1) → row 3, column 1 when compared to goal state.
Manhattan Distance=|2−3|+|3−1|=1+2=3
In this way when all the tiles are considered we get
3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18
Therefore $h_2(n) = 18$

***Example-02:* Bengaluru to Chennai.**

Imagine you're traveling from Bengaluru to Chennai, and your plan includes stopping at cities like Krishnagiri, Vellore, and Chengalpattu. Now suppose you're currently in Chittoor, which is not on your planned route.
In this case, $h_1(n)$ could just count how many cities you're off track from your original plan. It doesn't measure how far you are in kilometers or time — it just says you're not on the correct route.
So, $h_1(n)$ is a rough idea of how many things are not in the right place, or how far off your path is in terms of wrong steps.
Now, instead of counting wrong steps, $h_2(n)$ looks at the actual travel distance left to reach Chennai. For example, if you're in Vellore and Chennai is still 140 km away, then $h_2(n) = 140$. This is much more accurate because it tells you how much distance or cost remains, not just whether you're off the path.
$h_1(n)$ tells you how many wrong turns or places are off your ideal route, while, $h_2(n)$ tells you how much distance/time is left to reach the destination.

3.9 Effective Branching Factor(b*):

The effective branching factor is a way to measure how efficiently a search algorithm is working. It tells us, on average, how many choices (branches) the algorithm had to explore at each level of the search to find the goal. A lower number means the algorithm was smart and explored fewer unnecessary paths. A higher number means it checked many options and may not have been very efficient. A well designed heuristic would have a value of $b*$ close to 1, allowing fairly large problems to be solved at reasonable computational cost.
It uses the formula (approximate) given below.

$$N + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d$$

Where:

- N = total number of nodes generated (including the goal)
- d = depth of the solution (number of steps to goal)
- b* = effective branching factor (what we want to find)

Example with Bengaluru to Chennai:

Imagine you're driving from Bengaluru to Chennai, and at each junction, you have several route options — like going through Krishnagiri, Kolar, or Hosur. A smart traveler (like a good heuristic search) will choose the most promising route quickly, without checking all options. This leads to a low effective branching factor.
But if you don't have a good idea (poor heuristic), you might try many different roads before finding the right one, like checking all possible paths through Chittoor, Ambur, Vaniyambadi, etc. This would make the search slower, and the effective branching factor would be high.

Problem: Determine effective branching factor for the heuristic given below.
Bengaluru → Krishnagiri → Vellore → Chengalpattu → Chennai with 13 number of nodes.

Solution: Given N = 13, d = 4 (it took 4 steps from start to goal)
Using the equation

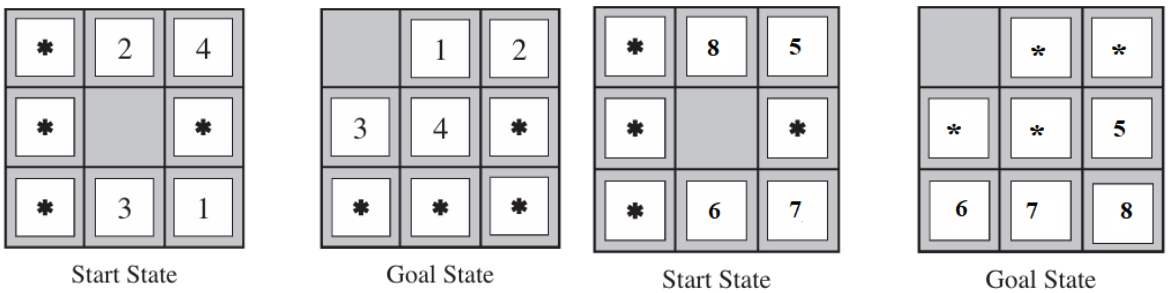
$$N + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d$$ where N = 13, d = 4 steps

$$N + 1 = 14 = 1 + b^* + (b^*)^2 + (b^*)^3 + (b^*)^4$$

From trial and error (feed equation in the calci and try for approximate value).
On trial and iteration we found b* = 1.5 which is very close to 14.
This means that, on average, the algorithm only had to consider 1.5 choices per step, which shows that the heuristic helped reduce unnecessary exploration. Without a heuristic, the branching factor might have been 3 or 4 (more paths per junction).

3.10 Pattern Database:

A pattern database is a smart way to improve the performance of search algorithms by precomputing exact solutions to smaller parts (or subproblems) of a larger problem. Instead of estimating how far a state is from the goal using basic heuristics, pattern databases store the actual number of steps needed to solve parts of the problem and use these as admissible heuristics.
Example: In the case of the 8-puzzle, shown in the following figure, the full puzzle is split into two smaller patterns—for example, one database for tiles {1,2,3,4} and another for tiles {5,6,7,8}.



Each pattern stores the minimum number of moves needed to get those tiles into the correct position, ignoring the others. During the actual search, the algorithm looks up the moves needed for each pattern based on the current state and adds them together to get a stronger heuristic estimate.
The total heuristic value is then calculated as:

$$h(n) = hp1(n) + hp2(n)$$

where:

- hp1(n) is the number of steps needed to solve the {1, 2, 3, 4} pattern.
- hp2(n) is the number of steps needed to solve the {5, 6, 7, 8} pattern.

This gives a more accurate and still admissible heuristic because it is based on real move counts. By using pattern databases, the search becomes faster and more efficient since the algorithm is guided by more precise information. So, instead of guessing or calculating during the search, it uses pre-solved mini-puzzles to guide the solution of the full puzzle intelligently.

Since both heuristics are based on actual move counts and don't overestimate, their sum is also admissible**,** meaning it still guarantees optimal solutions. Pattern databases are usually designed with non-overlapping tile sets. In case if hp1(n) + hp2(n) uses overlapping moves (i.e., both rely on the same blank tile moves), the combined heuristic might overestimate the cost and this violates admissibility**.** That means A* could no longer guarantee finding the optimal solution**.**

3.11 LOGICAL AGENTS

A logical agent is an intelligent system that makes decisions by applying rules of logic to a set of known facts. It stores knowledge about its environment in a structured form, usually using a knowledge base, and uses logical inference to deduce new information or choose actions. 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. Instead of reacting blindly, a logical agent reasons about what it knows to act smartly and adapt to different situations. This makes it more flexible, explainable, and capable of solving complex problems.

3.12 KNOWLEDGE- BASED AGENT:

A Knowledge-Based Agent is an intelligent system that makes decisions by using explicit knowledge about the world and applying logical reasoning to that knowledge. It can store facts, infer new information, and adapt to new situations using a central structure called a knowledge base (KB).

The KB is a repository of all the agent's knowledge in the form of statements or facts, typically written in a formal language. The agent uses two main operations:

1. *Tell* – to add new information to the KB.
2. *Ask* – to query the KB and infer conclusions using logical reasoning.

It uses three types of the following functions.

1.Make-Percept-Sentence

This function creates a logical sentence that says what the agent observed at a given time. *For example,* if the agent sees "it is raining" at time t = 5, the function constructs a sentence like: *Percept (ItIsRaining, 5).* This new information is then added to the agent's knowledge base using the TELL operation, allowing the agent to remember what it has observed.

2. Make-Action-Query

This function builds a logical question, asking the knowledge base:

"What should I do now, based on what I know?" For example, if the agent is at time t = 5, it might ask: ASK(KB, WhatAction(5))

The inference engine inside the agent then searches through its knowledge to determine the best action based on rules, previous experiences, and the current situation.

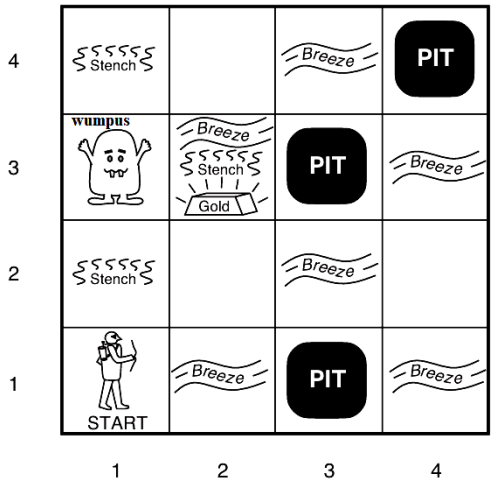
3. Make-Action-Sentence

After the agent chooses an action, this function creates a sentence that records the action taken, such as: Action(Do(OpenUmbrella), 5)

This sentence is then also added to the knowledge base with TELL, so the agent remembers what it did at each step. This is important for reasoning about consequences or explaining its behavior later.These steps together enable the agent to perceive, reason, and act intelligently, keeping track of all decisions and observations for future reasoning.

3.13 THE WUMPUS WORLD

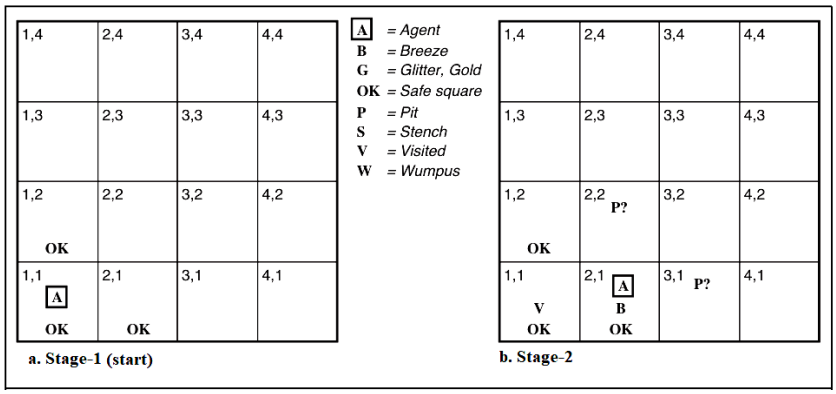
The Wumpus World is a classic artificial intelligence environment used to demonstrate how a knowledge-based agent can use logic to make smart decisions in an uncertain world. The environment is a 4x4 grid of rooms. Some rooms contain deadly pits, and one room contains a fearsome creature called the Wumpus, which can kill the agent as shown in the following figure.



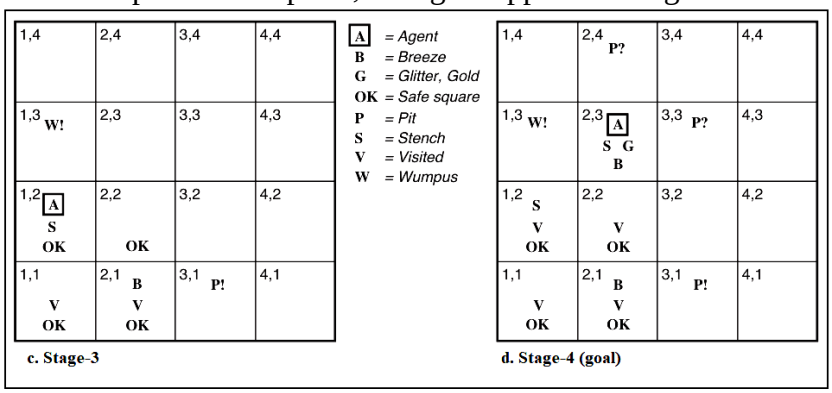
There's also a room with gold, which the agent must try to find and bring back safely. The agent starts in the bottom-left corner of the grid ([1,1]) and can move forward, turn left or right, grab the gold, shoot an arrow to kill the Wumpus, and climb out if it reaches the starting point with the gold. The challenge is that the agent doesn't know where the pits or Wumpus are initially, so it must use sensory information like breeze (near a pit), stench (near the Wumpus), glitter (near gold), bump (if it hits a wall), and scream (if the Wumpus dies) to infer what's around it. Using logic-based reasoning, the agent marks squares as safe or dangerous and decides where to go next

3.14 Wumpus World Agent Example:

In the Wumpus World, the agent starts at square [1,1], where it perceives no breeze and no stench. From this, it logically concludes that all adjacent squares—[1,2] and [2,1]—are safe. It first moves to [2,1], where it detects a breeze, indicating that a pit may exist in one of its neighboring squares, possibly [2,2] or [3,1]. Not yet certain where the pit is, it marks those squares as possibly dangerous.



Next, the agent explores [1,2], where again it senses no breeze or stench. Since [1,2] is adjacent to [2,2], the lack of breeze confirms that [2,2] cannot contain a pit and must be safe. Encouraged by this, the agent moves to [2,2], where it now perceives both a breeze and a stench. This implies that one or more of the neighboring squares—[2,3] or [3,2]—might contain a pit or the Wumpus. At this point, the agent applies the logical rule shown in Figure.



"If there is a breeze in a square, then one of its neighboring squares contains a pit." From its earlier observation that [2,1] had a breeze and now knowing that [2,2] is safe, it deduces that the pit must be in [3,1]. This reasoning allows the agent to mark [3,1] as unsafe and avoid it, helping the agent navigate more safely. By using such logical inference with each percept, the agent gradually builds a map of safe and unsafe squares, allowing it to move intelligently toward the gold [2,3] while avoiding hazards. In [2,3], the agent detects a glitter, so it should grab the gold and then return home. Note that in each case for which the agent draws a conclusion from the available in formation, that conclusion is guaranteed to be correct if the available information is correct. This is a fundamental property of logical reasoning.

3.15 LOGIC:

Logic introduces the foundational ideas behind logical representation and reasoning, which are central to building intelligent agents. It consists of the following components.

Syntax: Syntax refers to the rules that define how sentences or statements are formed in logic. It tells us what a valid sentence looks like, just like grammar rules in a language. For example, in propositional logic, a valid statement could be "x +y =4", but something like "x4y+=" would be syntactically incorrect.

Semantics: It tells us what a sentence or statement actually means, and whether it is true or false in a certain situation. It gives the meaning behind those sentences. It evaluates the truth of a sentence under given values. It's like asking: "Does this statement make sense and hold true in this situation?" For example, the semantics for arithmetic equation specifies that the statement "x + y =4" is true in a world where x is 2 and y is 2, but false in a world where x is 1 and y is 1.

Model: A model is a specific assignment of values (or facts) that makes a statement true. It's like a "world" where certain things are true.. For example, for the above equation " x+y = 4" if x=1 and y=3 it will be true, while if x=1 and y =2 it will be false. Logic uses models to check whether a sentence holds true under certain conditions.

Entailment (α ⊨ β *ie* "alpha entails beta"***):*** Entailment means that one sentence (α) logically leads to another (β). If α is true, then β must also be true in all models where α holds. This is written as α ⊨ β, showing a guaranteed logical connection between facts and conclusions. For example: If x+y = 4 and for x=1 then ⊨ y = 3

Inference Rules: Inference rules are logical steps or formulas used to derive new truths from existing facts. For example for the equation x + y = 4 if x =2 then using above y can be found. So, using logical reasoning, we inferred that y = 2.

Soundness: A reasoning system is sound if every conclusion it draws is actually true in the real world. This means it never produces false results when working with true premises.
Completeness: A logic system is complete if it can derive every statement that is logically true. In other words, it won't miss any valid conclusions that should logically follow from the given knowledge.
These concepts together help build intelligent systems that can reason correctly and reliably using logic.
Grounding & Learning: Grounding refers to how an agent's internal knowledge (its knowledge base or KB) is connected to the real world. Since the KB is just a set of sentences inside the agent's mind, a key question is: *How do we know that what the agent "knows" is actually true?* The simple answer is that the agent's sensors provide this connection.
For example, in the Wumpus World, when the agent smells something, its program creates a logical sentence like "there is a stench." Because the sensor detected it, that sentence is considered true in the real world. So, percepts (what the agent senses) form the base for truth in the knowledge base.
However, not all knowledge comes from direct sensing.For example, the rule that "a Wumpus causes a stench in neighboring squares" is a general belief, likely learned from past experiences. This type of knowledge is formed through a process called ***learning***, which is not always perfect.
Mistakes can happen—for example, maybe Wumpuses don't smell on leap years! So while the KB may not always be completely true, good learning methods help the agent build accurate and useful knowledge about the world.
In conclusion Grounding connects an agent's internal knowledge with the real world, mainly through sensors and perception. A grounded KB helps the agent reason more accurately and act intelligently in real situations.

3.16 PROPOSITIONAL LOGIC:

Propositional logic is the simplest form of logic used in AI. It deals with statements (propositions) that are either true or false, but it does not go into the internal structure of those statements. Each proposition is represented by a symbol like P, Q, or R, and logical connectives (like AND ∧, OR ∨, NOT ¬, IMPLIES →) are used to build more complex sentences.
For example, "It is raining" might be represented by R, and "The ground is wet" by W. A logical sentence like R → W means "If it is raining, then the ground is wet."
Propositional logic allows agents to store knowledge, make logical inferences, and decide what is true based on known facts. Some of the Reasoning patterns used in propositional logic are shown in the following table.

Symbol	Name	Meaning / Use
¬	NOT (Negation)	Reverses truth value: ¬P is true if P is false
∧	AND (Conjunction)	True if both P and Q are true
∨	OR (Disjunction)	True if at least one of P or Q is true
⊕ or XOR	Exclusive OR	True if exactly one of P or Q is true
→ or ⇒	IMPLIES (Conditional)	P ⇒ Q means if P is true, then Q is true
↔ or ⇔	BICONDITIONAL (IFF)	True if P and Q are both true or both false
⊨	ENTAILMENT	P ⊨ Q means P logically leads to Q
⊢	PROVABILITY	P ⊢ Q means Q can be derived from P
⊥	CONTRADICTION (Bottom)	Represents falsehood or inconsistency

Symbol	Name	Meaning / Use
⊤	TAUTOLOGY (Top)	Represents a statement that is always true

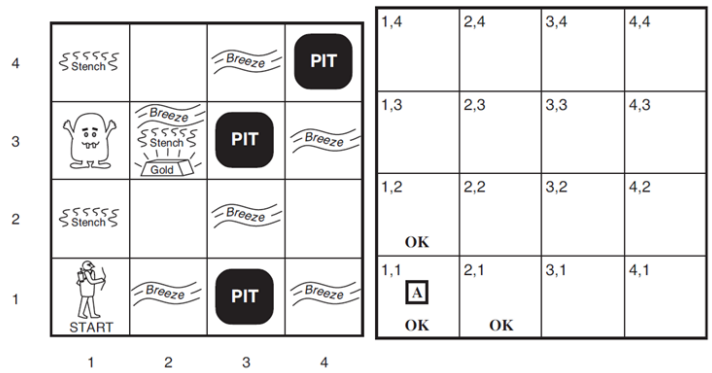
Example: Consider a vocabulary with only four propositions, A, B, C and D. How many models are there for the following sentences?

i. **B ∨ C**
ii. **¬A ∨ ¬B ∨ ¬C ∨ ¬D**
iii. **(A ⇒ B) ∧ A ∧ ¬B ∧ C ∧ D**

Solution:
(i) B ∨ C
This means: *"B OR C is true."*
 When is it false?
Only when both B and C are false.
B = 0, C = 0 → the only false case
All other combinations (15 out of 16) make it true.
Answer: 15 models
(ii) ¬A ∨ ¬B ∨ ¬C ∨ ¬D
This means: *"At least one of A, B, C, or D is false."*
 When is it false?
Only when all are true:
A = 1, B = 1, C = 1, D = 1
That's 1 model where it's false.
All other 15 models make the sentence true.
Answer: 15 models
iii) (A ⇒ B) ∧ A ∧ ¬B ∧ C ∧ D
We're asked: How many truth assignments (models) make this whole expression true?
Step-by-step:
We are saying:
A is true (A = T □)
B is false (¬B = T → B = F □)
C is true (C = T □)
D is true (D = T □)
A ⇒ B must be true
(This is the tricky part)
Focus on A ⇒ B (means: *"If A is true, then B must be true"*)
But we already said:
A = true
B = false
Let's test this:
A ⇒ B = true ⇒ false = □ false
So the condition (A ⇒ B) is false.
□ Final Result:
You are trying to say:
A is true
B is false
And (A ⇒ B) is true
But (A ⇒ B) is false in this case.
So you are trying to combine true and false using AND (∧), which results in false.
This statement is impossible to satisfy. There is no truth assignment where:
A is true
B is false
And A ⇒ B is also true
Answer: 0 models satisfy this expression

3.17 Syntax with Wumpus World

In propositional logic, syntax defines the rules for writing well-structured logical statements using specific symbols and operators. In the context of the Wumpus World shown in the fig, where each cell in the grid may contain a breeze, stench, pit, or Wumpus, we can assign propositional symbols to represent these conditions as stated below.



For example, B21 for "breeze in square [2,1]" and P31 for "pit in square [3,1]". Using logical operators, we can form complex but valid statements that express the relationships between these symbols.
For instance, the negation symbol ¬ is used to indicate that something is not true: ¬P21 means "there is no pit in square [2,1]".

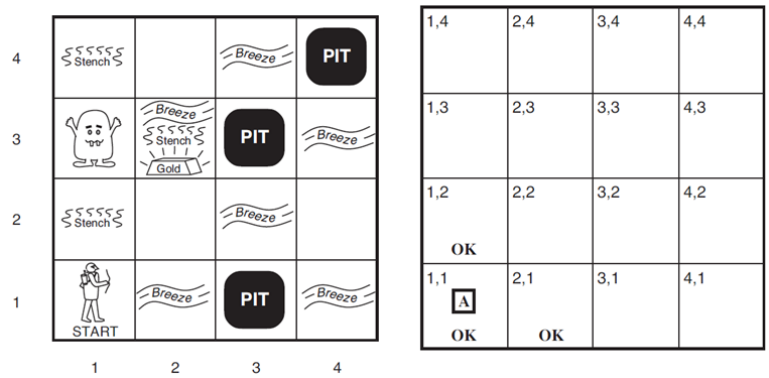
The conjunction operator ∧ (AND) allows us to combine two facts that are both true B23 ∧ S23 means "there is a breeze and a stench in square [2,3]".
The disjunction operator ∨ (OR) represents uncertainty or alternative possibilities: P33 ∨ P44 means "either [3,3] or [4,4] has a pit".
The implication symbol ⇒ (IF...THEN) is used to express a cause-effect relationship: B21 ⇒ (P31 ∨ P22) means "if there is a breeze in [2,1], then there must be a pit in one of the adjacent squares".
Lastly, the biconditional symbol ⇔ (IF AND ONLY IF) indicates two sides are logically equivalent: B21 ⇔ (P31 ∨ P22) says "a breeze exists in [2,1] if and only if there is a pit in [3,1] or [2,2]".
All these expressions are syntactically valid because they follow the formal rules of how symbols and operators can be combined. The agent relies on such syntactically correct statements to reason about its world logically. However, syntax only checks whether a sentence is correctly formed; it does not tell us if the sentence is true in the real environment which is determined by semantics.

3.18 Semantics with Wumpus World

Semantics in propositional logic deals with the meaning of sentences — that is, whether a sentence is true or false in a particular situation or interpretation. Each atomic proposition, like P or Q, can be assigned a truth value: either true (T) or false (F). The semantics of compound sentences formed using logical operators depend on the truth values of their parts. For example,
The sentence P ∧ Q (P AND Q) is true only when both P and Q are true. If either one is false, the whole sentence is false.
Similarly, P ∨ Q (P OR Q) is true if at least one of them is true.
¬P (NOT P) is true when P is false, and false when P is true.
The sentence P ⇒ Q (P IMPLIES Q) is only false when P is true but Q is false; in all other cases, it's true.
And P ⇔ Q (P if and only if Q) is true when both P and Q are either true or both false.
The above five logical connectives are illustrated in the following truth table.

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

So, semantics gives a clear truth-based interpretation to the syntax (structure) of logical sentences. In the context of intelligent agents, semantics helps the agent evaluate which logical sentences match the actual state of the world, so it can make correct decisions. For example, if P represents "It is raining" and Q represents "The ground is wet", then understanding the truth of P ⇒ Q helps the agent relate its observations to real-world outcomes.
Wumpus World: In the Wumpus World, semantics tells us whether a logical sentence about the world is actually true or false based on the agent's observations. For instance, in the grid, the agent starts at cell [1,1], where it does not perceive any breeze or stench. Based on this, the sentence ¬B11 ∧ ¬S11 (no breeze AND no stench in [1,1]) is true, as confirmed by the environment as shown in the following figure.



Semantically, this means adjacent cells like [1,2] and [2,1] cannot have pits (since breezes are only felt next to pits). Therefore, the sentence B11 ⇒ (P12 ∨ P21) is false, because B11 is false and so the implication holds by default — but the real meaning here is that no pit is in the adjacent squares.

As the agent moves to [2,1], it perceives a breeze, so B21 is true. That gives semantic weight to the rule B21 ⇒ (P11 ∨ P22 ∨ P31), which says that if there's a breeze in [2,1], then there must be a pit in one of its neighboring squares. Since [1,1] has already been declared safe, the agent can conclude the pit must be in either [2,2] or [3,1], although it doesn't yet know which. This is reflected in the right-side grid, where [2,1] is marked OK, but adjacent unknown squares are not confirmed yet. The agent's understanding of which logical sentences are true evolves as it moves, and it uses semantics — the actual content of its percepts — to judge the truth value of each logical proposition in the environment.

QUESTIONS:

5-Mark Questions

1. Explain the concept of heuristic function with a suitable example.
2. Differentiate between Greedy Best-First Search and A* Search.
3. Define and distinguish between admissibility and consistency in the context of A* Search.
4. What is the role of heuristic functions in search strategies? Briefly explain $h_1(n)$ and $h_2(n)$ with an example.
5. List and explain the basic components of logic used in AI.
6. What is the effective branching factor? How is it calculated? Explain with an example.
7. Describe the structure and role of a Knowledge-Based Agent. What are the three key functions it performs?
8. Write short notes on: (i) Recursive Best-First Search (RBFS), (ii) Simplified Memory-Bounded A* (SMA)
9. Explain with an example how a logical agent deduces safe and unsafe squares in the Wumpus World.
10. Define pattern databases. How do they improve the performance of search algorithms in AI?

10-Mark Questions

1. Apply Greedy Best-First Search on the given graph and explain the path selection with reasoning at each step. (Based on Numerical-01 in Section 3.2)
2. With the help of an example, explain the working of A Search algorithm. Include open and closed lists at each step. (Based on Numerical-02 in Section 3.3)
3. What are the limitations of Greedy Best-First Search? How does A Search overcome them?
4. Illustrate the concept of heuristic admissibility and consistency using suitable numerical examples.
5. Explain how knowledge is represented and used in a Knowledge-Based Agent with reference to Wumpus World.

6. Explain the concepts of syntax, semantics, and entailment in propositional logic with appropriate examples.
7. Compare and contrast $h_1(n)$ and $h_2(n)$ using the 8-puzzle problem. In what scenarios is $h_2(n)$ preferred over $h_1(n)$?
8. Describe the Wumpus World environment. How does a logical agent use percepts and reasoning to make decisions in such a world?
9. Discuss the significance of pattern databases in heuristic search. How does overlapping tile usage affect admissibility?
10. Demonstrate how propositional logic and reasoning patterns are applied in AI problem solving. Give suitable examples.

BCS515B

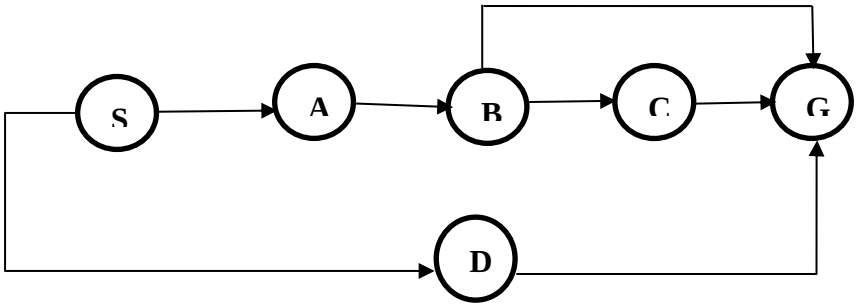
Model Question Paper with effect from 2023-24 (CBCS Scheme)

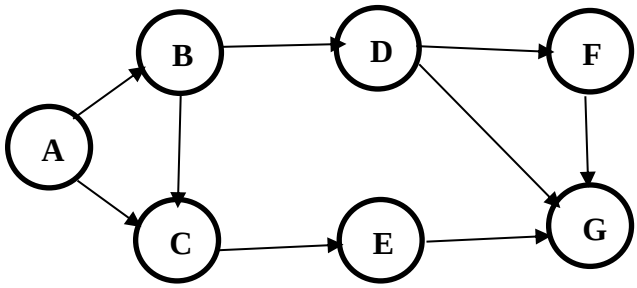
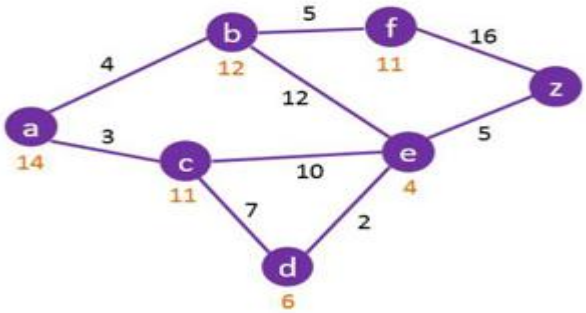
USN | | | | | | | | | | |

Fifth Semester B.E. Degree Examination
Artificial Intelligence

TIME: 03 Hours **Max. Marks: 100**

Note: Answer any FIVE full questions, choosing ONE full question from each module.

Q. No.		Questions	Marks	BL
		Module 1		
1	a.	Define Artificial Intelligence. Explain the foundations of AI in detail.	10	CL2
	b.	Discuss the PEAS specification of Biometric Authentication System	10	CL3
		OR		
2	a.	Differentiate: i. Fully observable vs partially observable ii. Single agent vs Multiagent iii. Deterministic vs Stochastic iv. Static vs Dynamic	10	CL2
	b.	Give PEAS specification for Automated Taxi Driver	10	CL3
		Module 2		
3	a.	Explain five components and well-defined problem. Consider an 8-puzzle problem as an example.	10	CL2
	b.	In detail elaborate with neat diagram, the state space for the vacuum world. Links denote actions: L = Left,R= Right,S=Suck.	10	CL3
		OR		
4	a.	For the graph given below apply BFS and DFS Search algorithm.  	10	CL3
	b.	Explain Depth Limited Search and Iterative Deeping DFS Search with an example.	10	CL2
		Module 3		
5	a.	In the below graph, find the path from A to G. Using Greedy Best First search and A* search algorithm. The values in the table represent heuristic values of reaching the goal node G pass current node.	10	CL3

		 A 5 B 6 C 4 D 3 E 3 F 1 G 0		
	b.	Explain the Knowledge–based agents with an algorithm and explain the TELL and ASK functions used for same.	10	CL2
OR				
6	a.	Apply the A* search to find the solution path from a to z. Heuristics are with nodes, and cost is with edges. Write all steps as well as open and closed lists for full marks 	10	CL3
	b.	Considering Wumpus world example give the characteristics and simple knowledge base using propositional logic and **Prove that Wumpus is in the room (1, 3).**	10	CL3
Module 4				
7	a.	List the drawbacks of Propositional Logic. Explain the Syntax and Semantics in First order Logic.	10	CL2
	b.	Define Universal and Existential Instantiation and give examples for both. Prove the following using Forward chaining: "As per the law, it is a crime for an American to sell weapons to hostile nations. Country E, an enemy of America, has some missiles, and all the missiles were sold to it by Solan, who is an American citizen." Prove that "Solan s criminal."	10	CL3
OR				
8	a.	Explain the following with respect to the first order logic: i. Assertions and Queries in First order logic ii. The kinship domain iii. Numbers, sets and lists.	10	CL2
	b.	Considering the digital circuit example illustrate the seven steps in knowledge engineering process.	10	CL2
Module 5				
9	a.	Explain Backward Chaining algorithm with an example.	10	CL2

	b.	Define Classical Planning. With the blocks world example explain the same in detail.	10	CL2
OR				
10	a.	Write appropriate quantifiers for the following (i) Some students read well (ii) Some students like some books (iii) Some students like all books (iv) All students like some books (v) All students like no books Explain the concept of Resolution in First Order Logic with appropriate procedure.	10	CL2
	b.	Explain the two approaches to searching for a plan in detail.	10	CL2

Cognitive Levels of Bloom's Taxonomy

No.	CL1	CL2	CL3	CL4	CL5	CL6
Level	Remember	Understand	Apply	Analyze	Evaluate	Create

CBCS SCHEME

EPCET LIBRARY BANGALORE-49

USN | | | | | | | | | | |

BCS515B

Fifth Semester B.E./B.Tech. Degree Examination, Dec.2024/Jan.2025
Artificial Intelligence

Time: 3 hrs. Max. Marks: 100

Note:* 1. *Answer any FIVE full questions, choosing ONE full question from each module.
2. M : Marks , L: Bloom's level , C: Course outcomes.

		Module – 1	M	L	C
Q.1	a.	Define the following : i) Intelligence ii) Artificial Intelligence iii) Agent iv) Rationality v) Logical reasoning.	5	L2	CO1
	b.	Examine the AI literature to discover whether the following tasks can currently be solved by computers. i) Playing a decent game of table tennis (ping-pong) ii) Discovering and proving new mathematical theorems iii) Giving competent legal advice in a specialized area of law iv) Performing a complex a surgical operation.	8	L2	CO1
	c.	Implement a simple reflex agent for the vacuum environment. Run the environment with this agent for all possible initial dirt configurations and agent locations. Record the performance score for each configuration and the overall score.	7	L3	CO1
		OR			
Q.2	a.	Is AI a science, or is it engineering or neither or both? Explain.	5	L2	CO1
	b.	Write pseudocode agent programs for the goal based and utility based agents.	8	L1	CO1
	c.	For each the following activities give a PEAS description. i) Playing a tennis match ii) Performing a high jump iii) Bidding on an item in an auction.	7	L1	CO1
		Module – 2			
Q.3	a.	Explain why problem formulation must follow goal transformation.	5	L1	CO1
	b.	Give complete problem formulation for each of the following choose a formulation that is precise enough to be implemented. i) Using only four colors, you have to color a planar graph in such a way that no two adjacent regions have the same color. ii) A 3 – foot – tall monkey is in a room where some bananas are suspended from the 8-foot ceiling. He would like to get the bananas. The room contains two stackable, moveable, climbable 3-foot high crates.	8	L2	CO2
	c.	Prove each of the following statements or given counter example : i) Breadth – first search is a special case of uniform – cost search. ii) Uniform – cost search is a special case of A^* search.	7	L2	CO2

VTU-22-01-2025 01:08:23pm - EP - EP - EP - EP
22-01-2025 01:29:37pm - EP - EP - EP - EP - EP - EP - EP - EP - EP - EP - EP - EP

		OR			
Q.4	**a.**	Define the following terms with example. i) State space ii) Search node iii) Transition model iv) Branching factor.	**8**	**L2**	**CO2**
	b.	Show that the 8-puzzle states are divided in to two disjoint sets, such that any state is reachable from any other state in the same set, while no state is reachable from any state in the other set. Devise a procedure to decide which set a given state is in and explain why this is useful for generating random state.	**7**	**L2**	**CO2**
	c.	Describe a state space in which iterative deepening search performs much worse than depth first search for example, $O(n^2)$Vs $O(n)$).	**5**	**L2**	**CO2**
		Module – 3			
Q.5	**a.**	Devise a state space in which A^* using GRAPH-SEARCH returns a suboptimal solution with h(n) function that is admissible but inconsistent.	**7**	**L2**	**CO3**
	b.	Which of the following are correct? i) $(A \vee B) \wedge (\neg C \vee \neg D \vee E) F (A \vee B)$ ii) $(A \vee B) \wedge (\neg C \vee \neg D \vee E) F (A \vee B) \wedge (\neg D \vee E)$ iii) $(A \vee B) \wedge \neg(A \Rightarrow B)$ is satistiable iv) $(A \Leftrightarrow B) \Leftrightarrow C$ has the same number of models as $(A \Leftrightarrow B)$	**8**	**L1**	**CO3**
	c.	Consider a vocabulary with only four propositions, A, B, C and D. How many models are there for the following sentences? i) $B \vee C$ ii) $\neg A \vee \neg B \vee \neg C \vee \neg D$ iii) $(A \Rightarrow B) \wedge A \wedge \neg B \wedge C \wedge D$.	**5**	**L1**	**CO3**
		OR			
Q.6	**a.**	Prove that if a heuristic is consistent, it must be admissible. Construct an admissible heuristic that is not consistent.	**8**	**L1**	**CO3**
	b.	Prove each of the following assertions : i) $\alpha \equiv \beta$ if and only if the sentence $(\alpha \Leftrightarrow \beta)$ is valid ii) $\alpha \neq \beta$ if and only if the sentence $\alpha \wedge \neg\beta)$ is unsatisfiable.	**7**	**L1**	**CO3**
	c.	Prove, or find a counter example to each of the following assertions. i) If $\alpha \neq (\beta \wedge \gamma)$ then $\alpha \neq \beta$ and $\alpha \neq \gamma$. ii) If $\alpha \neq (\beta \vee \gamma)$ then $\alpha \neq \beta$ and $\alpha \neq \gamma$ (or) both	**5**	**L1**	**CO3**
		Module – 4			
Q.7	**a.**	Which of the following are valid necessary true sentences? i) $(\exists x\, x = x) \Rightarrow (\forall y \exists z\, y = z)$ ii) $\forall x P(x) \vee \neg p(x)$ iii) $\forall x$ smart(x) $\vee (x = x)$	**7**	**L1**	**CO4**
	b.	Prove that universal Instantiation is sound that existential instanticition produces an inferentially equivalent knowledge base.	**5**	**L1**	**CO4**

VTU - 22-01-2025 01:08:23pm - EP - EP - EP
22-01-2025 01:29:37pm - EP - EP - EP - EP - EP

	c.	Write down logical representations for the following sentences, suitable for use with generalized modulus ponens : i) Horses, cows and pigs are mammals ii) Bluebeard is Charlie's parent iii) Offspring and parent are inverse relations	8	L1	CO4
		OR			
Q.8	a.	Consider a knowledge base containing just two sentence ; P(a) and P(b) does this knowledge base entail $\forall x\, P(x)$? Explain your answer interms of models.	5	L2	CO4
	b.	Suppose a knowledge base contains just one sentence, $\exists x\, AsHighAs(x, Everest)$ which of the following are legitimate results of applying existential instantiation? i) AsHighAs(Kilimanjaro, Everest) ii) AsHighAs(Kilimanjaro, Everest) $\wedge$ AsHighAs (Benvevis, Everest)	8	L2	CO4
	c.	Explain how to write any 3-SAT problem of arbitrary size using a single first order definite clause and no more than 30 ground facts.	7	L2	CO4
		Module – 5			
Q.9	a.	i) Give a backward chaining proof of the sentence $7 \leq 3 + 9$. Show only the steps that leads to success ii) Give a forward chaining proof of the sentence $7 \leq 3 + 9$. Show only the steps that leads to success.	8	L1	CO5
	b.	Describe the differences and similarities between problem solving and planning.	5	L2	CO5
	c.	Prove that backward search with PDDL problems is complete.	7	L1	CO5
		OR			
Q.10	a.	The following prolog code defines a predicate P P(x, [x|y]), P(x, [y|z]) :- P(x, z) i) Show proof trees and solutions for the queries P(A, [2, 1, 3]) and P(z,[1, A, 3]) ii) What standard list operation does P represent?	8	L1	CO5
	b.	Explain why dropping negative effects from every action schema in a planning problem results in a relaxed problems.	5	L2	CO5
	c.	Prove the following assertions about planning graphs : i) A literal that does not appear in the final level of the graph connot be achieved. ii) The level cost of a literal in a serial graph is no greater than the actual cost of an optimal plan for achieving it.	7	L1	CO5

* * * * *

MTU-22-01-2025 01:08:23pm
EP - EP - EP - EP - EP - EP - EP - EP
22-01-2025 01:29:37pm
EP - EP - EP - EP - EP - EP - EP - EP

CBCS SCHEME

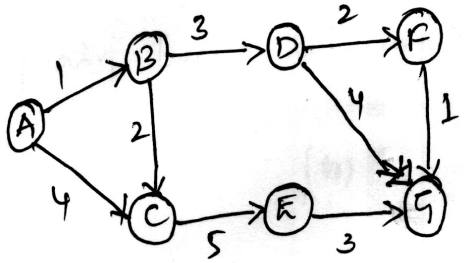
USN | | | | | | | | | | |

BAD402

Fourth Semester B.E./B.Tech. Degree Examination, June/July 2024
Artificial Intelligence

Time: 3 hrs. Max. Marks: 100

Note:* 1. *Answer any FIVE full questions, choosing ONE full question from each module.
2. M : Marks , L: Bloom's level , C: Course outcomes.

		Module – 1	**M**	**L**	**C**
Q.1	**a.**	Define Artificial Intelligence. Explain the foundation of AI in detail.	**10**	**L1**	**CO1**
	b.	Explain all four different approaches to AI in detail.	**10**	**L1**	**CO1**
		OR			
Q.2	**a.**	Give PEAS specification for : i) Automated taxi driver ii) Medical diagnostic system.	**10**	**L1**	**CO1**
	b.	Differentiation : i) Fully observable Vs partially observation ii) Single agent Vs Multiagent iii) Deterministic Vs stochastic iv) Static Vs Dynamic.	**10**	**L1**	**CO1**
		Module – 2			
Q.3	**a.**	Explain five components and well defined problem. Consider an 8-puzzle problem as an example and explain.	**10**	**L2**	**CO2**
	b.	Discuss in detail in Infrastructure for search algorithm.	**10**	**L2**	**CO2**
		OR			
Q.4	**a.**	Write an algorithm for Breadth – first search and explain with an example.	**10**	**L2**	**CO2**
	b.	Explain Depth first search techniques in detail.	**10**	**L2**	**CO2**
		Module – 3			
Q.5	**a.**	Explain the A* search to minimize the total estimated cost.	**10**	**L3**	**CO3**
	b.	Write an algorithm for hill climbing search and explain in detail.	**10**	**L3**	**CO3**
		OR			
Q.6	**a.**	In the below graph, find the path from A to G. Using Greedy Best First search and A* search algorithm. The values in the table represent heuristic values of reaching the goal node G pass current node.  Fig Q6(a)	**10**	**L3**	**CO3**

A	5
B	6
C	4
D	3
E	3
F	1
G	0

VTU Library - June/July 2024

	b.	Explain the syntax and semantion of propositional logic.	**10**	**L3**	**CO3**
		Module – 4			
Q.7	**a.**	Explain the syntax and semantics of the first order logic.	**10**	**L2**	**CO2**
	b.	Explain the following with respect to the first order logic i) Assertions and Queries in first order logic ii) The Kinship domain iii) Numbers, sets and lists.	**10**	**L2**	**CO2**
		OR			
Q.8	**a.**	Explain unification and lifting in detail.	**10**	**L3**	**CO4**
	b.	Explain Forward chaining algorithm with an example.	**10**	**L3**	**CO4**
		Module – 5			
Q.9	**a.**	Explain basic probability Notation in detail.	**10**	**L3**	**CO5**
	b.	Explain Baye's rule and its use in detail.	**10**	**L3**	**CO5**
		OR			
Q.10	**a.**	Explain Independence in Quantifying uncertainty with example.	**10**	**L3**	**CO5**
	b.	Explain knowledge Acquiting in detail.	**10**	**L3**	**CO5**

*** * * * ***

VTU Library - Jan/Jul 2024