B.E/B.TECH- Internal Assessment – I

Academic Year 2024-2025(ODD SEMESTER)

Third Semester

Artificial Intelligence and Machine Learning

23AMT201- FOUNDATIONS OF ARTIFICIAL INTELLIGENCE

PART A

1. Mention the problem solving techniques in AI.

AI problem-solving techniques include:

- Search Algorithms
- Optimization
- Logic-Based
- Machine Learning
- Knowledge-Based
- Heuristics
- Neural Networks
- Game Theory

2. Distinguish between informed and uninformed search strategies

Informed search uses heuristics to guide the search toward the goal efficiently (e.g., A*), while uninformed search explores blindly without any extra knowledge beyond the problem definition (e.g., BFS, DFS). Informed is faster but more complex, uninformed is simpler but can be slower.

3. Define constraint satisfaction

A constraint satisfaction problem (CSP) is a mathematical question where the goal is to find a solution that satisfies a set of constraints or conditions. In CSP, you are typically provided with:

- Variables
- Domains
- Constraints

4. What do you mean by logical agents?

Logical agents are artificial intelligence systems that use formal logic to represent knowledge and reason about it. They apply logical statements to draw conclusions, make decisions, and take actions based on their understanding of the world. These agents are capable of updating their knowledge as new information becomes available and are commonly used in areas like automated reasoning, planning, and natural language processing.

5. Justify the need of inferences.

Inferences are crucial because they help us understand hidden meanings, fill gaps in knowledge, make decisions with incomplete data, and reason efficiently without needing all the facts upfront.

PART- B

6. (a) Discuss in detail about the structure of different agents in artificial intelligence

In Artificial Intelligence (AI), agents are entities capable of perceiving their environment through sensors and acting upon it using actuators to achieve specific goals. AI agents can be designed to operate in a wide range of environments, from simple static ones to highly dynamic and complex settings. Based on their capabilities and design, AI agents can be classified into several types, each with its unique structure and behavior. Below is a detailed discussion of the different types of agents in AI:

1. Simple Reflex Agents

A simple reflex agent selects actions based solely on the current state of the environment, ignoring the history of previous states. These agents follow a condition-action (or rule-based) approach where the decision is made according to if-then rules.

Structure:

Perception: The agent perceives the environment.

Action Selection: It applies a set of condition-action rules (like "if condition, then action") to determine the action.

No Memory or Learning: They don't keep track of past states and are incapable of learning.

Example:

A thermostat that turns on a heater if the temperature falls below a certain threshold.

Limitations:

Inefficient in complex or dynamic environments because they only react to the current percept, without any consideration of the history or planning.

2. Model-Based Reflex Agents

A model-based reflex agent is an improvement over the simple reflex agent. It maintains some internal state that tracks aspects of the world that cannot be observed directly at every moment.

Structure:

Internal Model: It maintains an internal model of the world to track unseen or implicit aspects of the environment.

State Update: The agent updates its internal state using the current percept and the previous state.

Rules or Conditions: Like simple reflex agents, it uses condition-action rules but considers both current perceptions and the internal state.

Action Selection: Decides on the appropriate action based on the current state and internal model.

Example:

A robot vacuum cleaner that tracks where it has already cleaned, allowing it to make decisions based on both the current state and previous information.

Advantages:

More effective in partially observable environments because the internal model can help account for unseen changes.

3. Goal-Based Agents

A goal-based agent considers not just the current state but also aims to achieve specific goals. It uses information about the environment to evaluate potential actions based on how likely they are to achieve the desired goal.

Structure:

Goal Information: The agent is provided with one or more goals.

Planning: It searches through possible actions or states to find a sequence that will lead to the goal.

Internal State: It maintains knowledge about the environment to help guide the planning process.

Example:

A navigation system that computes a route to reach a desired destination by evaluating possible paths and choosing the one that will achieve the goal.

Advantages:

Provides more flexible behavior than reflex agents because it involves forward-looking planning to achieve goals.

Limitations:

May require significant computational resources if there are many possible actions or goals to evaluate.

4. Utility-Based Agents

A utility-based agent goes beyond goal-based agents by not only aiming to achieve goals but also considering the best way to achieve those goals. It assigns a utility value to different outcomes and selects actions that maximize expected utility.

Structure:

Utility Function: The agent uses a utility function to measure the "goodness" or desirability of different states or actions.

Decision Making: It evaluates actions based on how much utility each action brings toward achieving a goal.

Optimization: The agent selects the action that maximizes the utility or expected outcome, considering various factors like cost, time, and risk.

Example:

An autonomous vehicle that selects a route not just based on reaching a destination, but also factors in traffic, road conditions, and fuel efficiency to choose the optimal path.

Advantages:

More flexible than goal-based agents because it allows the agent to weigh trade-offs and make decisions that balance multiple objectives.

Limitations:

Requires a well-defined utility function, which can be difficult to define in complex environments.

(OR)

(b) Analyze how breadth first and depth first search algorithm to minimise the total estimated cost

Breadth-First Search (BFS) and Depth-First Search (DFS) are two classic search algorithms, but neither is directly designed to minimize the total estimated cost in problems where path cost matters. Instead, these algorithms are often used in problems where all step costs are equal. However, they can be adapted or combined with other techniques to minimize total estimated costs. Here’s a

breakdown of how BFS and DFS work, and how they relate to cost minimization:

1. Breadth-First Search (BFS)

Overview:

BFS explores all nodes at the present depth level before moving on to nodes at the next depth level. This guarantees finding the shortest path in an unweighted graph (i.e., all edges have the same cost), but not in a weighted graph.

How BFS Works:

Start from the initial node.

Explore all adjacent nodes.

Move to the next level and repeat until the goal is found.

Cost Minimization in BFS:

BFS is optimal for unweighted graphs because it explores nodes level by level. In such cases, the total cost is proportional to the depth of the node, and BFS guarantees finding the shallowest solution (i.e., the one with the least number of steps).

In weighted graphs, BFS cannot directly minimize cost, as it treats all edges as having equal weight. To address this, we use Uniform-Cost Search (UCS), which is essentially BFS but prioritizes nodes based on path cost rather than depth. UCS is optimal and complete, ensuring that the least-cost solution is found in weighted graphs.

2. Depth-First Search (DFS)

Overview:

DFS explores a path from the start node to the deepest node before backtracking. DFS does not guarantee the shortest path, and it can get stuck in deep, unproductive branches. However, it uses less memory compared to BFS, as it only needs to store nodes on the current path.

How DFS Works:

Start from the initial node.

Explore as far down a branch as possible before backtracking.

Continue exploring new branches until the goal is found.

Cost Minimization in DFS:

DFS is not optimal: It does not minimize cost because it explores paths without considering their lengths or costs.

Limited DFS (or Iterative Deepening DFS) can help in some cases by progressively increasing the depth limit. This way, it behaves like BFS and can find the shallowest solution, but it's still not ideal for minimizing costs in weighted graphs.

7. (a) Explain in detail about first order logic with prepositional logic

First-order logic (FOL) and propositional logic are both formal systems used in mathematical logic and computer science for representing and reasoning about knowledge, but they differ in expressiveness and structure. Let's break down each and then compare them.

1. Propositional Logic (PL)

Propositional logic is the simplest form of logic that deals with propositions or statements that are either true or false. It involves:

Atomic propositions: The most basic statements, which are indivisible and can either be true or false. For example, P, Q, and R could represent statements like "It is raining" or "The sky is blue."

Logical connectives: Used to combine or modify propositions. Common connectives include:

AND (∧): P ∧ Q (both P and Q must be true)

OR (∨): P ∨ Q (either P or Q, or both, must be true)

NOT (¬): ¬P (P is false)

Implication (→): P → Q (if P is true, then Q must be true)

Biconditional (↔): P ↔ Q (P is true if and only if Q is true)

Example of Propositional Logic:

P: "It is raining."

Q: "The ground is wet."

Using these propositions, we can form a statement like:

P → Q (If it is raining, then the ground is wet).

However, propositional logic is limited. It cannot express relationships between objects, quantify over objects, or represent properties of objects (like "all humans are mortal"). To handle more complex reasoning, we need first-order logic.

2. First-Order Logic (FOL)

First-order logic (also called predicate logic) extends propositional logic by introducing:

Objects: Entities in the domain of discourse (e.g., specific people, animals, numbers).

Predicates: Functions that describe properties or relations between objects. Predicates return true or false.

Example: Likes(John, Pizza) means "John likes pizza."

Quantifiers: These allow FOL to express statements about "some" or "all" objects in a domain:

Universal quantifier (∀): Represents "for all". For example, ∀x (Human(x) → Mortal(x)) means "For all x, if x is a human, then x is mortal."

Existential quantifier (∃): Represents "there exists". For example, ∃x (Cat(x) ∧ Black(x)) means "There exists an x such that x is a black cat."

Syntax of First-Order Logic

Terms: Objects in the domain (constants, variables, or functions).

Constants: Represent specific objects (e.g., John, Pizza).

Variables: Represent general objects (e.g., x, y, z).

Functions: Return a single object (e.g., MotherOf(x) represents the mother of x).

Atomic Formulas: Apply a predicate to a set of terms, such as Likes(John, Pizza) or Loves(x, y).

Complex Formulas: Combine atomic formulas with logical connectives and quantifiers, like ∀x ∃y (Loves(x, y)) (For every person x, there exists a person y such that x loves y).

Example of First-Order Logic:

Predicate: Loves(x, y) could represent "x loves y".

Quantified statement:

∀x ∃y (Loves(x, y)): "For every person x, there is a person y whom x loves."

∃x ∀y (Loves(x, y)): "There is a person x who loves every person y."

Why First-Order Logic is More Expressive:

Propositional logic cannot handle statements about individuals, their properties, or relationships between individuals. For example:

Propositional logic might represent "John loves Mary" and "John loves pizza" as two unrelated atomic statements P and Q.

First-order logic allows a deeper analysis by representing the relationship: Loves(John, Mary) and Loves(John, Pizza).

(OR)

(b) Peter can play cricket. Peter cannot play tennis. He can play tennis and cricket .He can play tennis and badminton .If Peter can play tennis then he can play badminton .He can play tennis if and only if he can play badminton. Derive implication statement in terms of proportional logic with truth table

Propositional Variables:

Let C = "Peter can play cricket"

Let T = "Peter can play tennis"

Let B = "Peter can play badminton"

Extracted Statements:

"Peter can play cricket" →

C

C

"Peter cannot play tennis" →

¬

T

¬T

"He can play tennis and cricket" →

T

∧

C

T∧C

"He can play tennis and badminton" →

T

∧

B

T∧B

"If Peter can play tennis, then he can play badminton" →

T

→

B

T→B

"He can play tennis if and only if he can play badminton" →

T

↔

B

T↔B

Implication to Derive:

We need to express all these conditions as a single logical implication statement.

From the last two conditions:

T

$\rightarrow$

B

T→B (If Peter can play tennis, he can play badminton)

T

$\leftrightarrow$

B

T↔B (He can play tennis if and only if he can play badminton)

This suggests that playing tennis and badminton are logically equivalent. Thus, the implication statement combining these conditions can be written as:

Truth Table:

We will now construct a truth table for

T

$\leftrightarrow$

B

T↔B along with the other important expressions

T

$\rightarrow$

B

T→B and

T

$\wedge$

B

T∧B, considering all possible combinations of truth values for

T

T,

B

B, and

C

C.

T	B	C	$\neg T$	$T \wedge C$	$T \wedge B$	$T \rightarrow B$	$T \leftrightarrow B$
T	T	T	F	T	T	T	T
T	T	F	F	F	T	T	T
T	F	T	F	T	F	F	F
T	F	F	F	F	F	F	F
F	T	T	T	F	F	T	F
F	T	F	T	F	F	T	F
F	F	T	T	F	F	T	T
F	F	F	T	F	F	T	T

8. (a) Apply the steps involved in search through problems face for 8 queens and explain in detail

The 8 Queens problem is a classic example of a constraint satisfaction problem (CSP), where the objective is to place 8 queens on a chessboard such that no two queens threaten each other. A queen can attack another queen if they are in the same row, column, or diagonal. The goal is to find all valid configurations for placing 8 queens.

The problem can be solved using several search strategies, including backtracking and depth-first search (DFS). Let's break down the steps involved in searching through this problem in detail.

Problem Statement:

We need to place 8 queens on an 8x8 chessboard so that no two queens are on the same row, column, or diagonal.

Step 1: Problem Representation

Search Space: The search space consists of all possible ways to place queens on the board. Since a queen can be placed in any row and column, the total number of potential placements is

8

!

=

8

×

7

×

6

×

.

.

.

×

1

=

40

,

320

8!=8×7×6×...×1=40,320 possible configurations. However, most of these configurations will violate the attack constraints.

Constraints:

No two queens should be in the same row.

No two queens should be in the same column.

No two queens should be on the same diagonal.

Step 2: Define the State

A state in this problem is defined by a partial arrangement of queens on the board. At each step, a queen is placed on one row in a valid column. The algorithm must ensure that the new queen does not attack any of the previously placed queens.

We can represent the state as a list of column numbers where the queens are placed in each row. For example, if the list is [4, 6, 1, 5], this means:

Queen in row 1 is in column 4,

Queen in row 2 is in column 6,

Queen in row 3 is in column 1,

Queen in row 4 is in column 5.

Step 3: Search Strategy

The most commonly used search strategy for the 8 Queens problem is Backtracking. This is a type of depth-first search (DFS) that explores all possibilities and backtracks when a conflict is found.

3.1 Backtracking Approach:

Start with an empty board (no queens placed).

Place a queen in the first row in the first column.

Move to the next row and try to place a queen in a column where it does not conflict with the previously placed queens (no two queens in the same column or diagonals).

If a valid position is found, move to the next row and repeat step 3.

If no valid position is found in a row, backtrack to the previous row and try the next column for the previous queen.

Continue this process until queens have been placed in all 8 rows

Step 4: Conflict Checking

Each time we place a queen, we need to check whether it violates any of the constraints:

Same Column: No two queens can be in the same column. This is easily ensured by maintaining a record of which columns have been occupied.

Same Diagonal: Two queens are on the same diagonal if the difference between their row indices equals the difference between their column indices.

Step 5: Backtracking Algorithm

Here is the high-level structure of the backtracking algorithm:

Recursive Function:

Define a function solve(row) that tries to place a queen in a valid column of the current row.

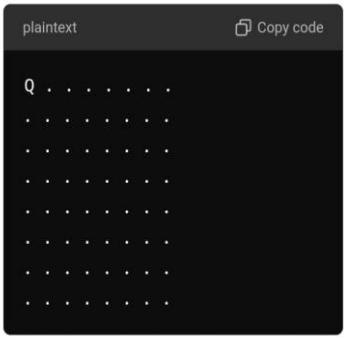
If row == 8 (all queens placed), print the solution.

Otherwise, try placing a queen in each column of the current row and check for conflicts.

The base case occurs when 8 queens have been placed, meaning a valid solution has been found.

Step 1: Initial Board

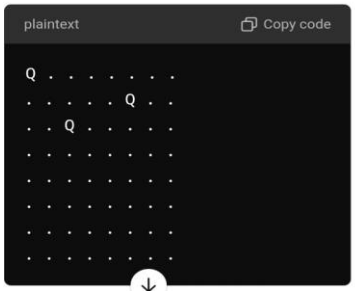
Step 2: Place the First Queen



Step 3: Place the Second Queen

Step 4: Continue Placing Queens

If we continue placing queens while respecting the rules, we may reach a point where placing the next queen leads to a conflict:

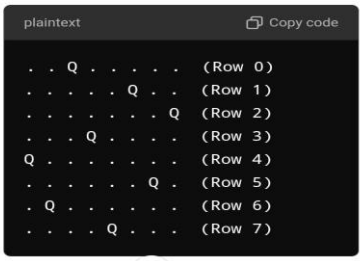


Step 5: Backtrack

When a conflict is encountered (like in the example above), we backtrack to the previous queen and try a different position for it:

Final Solution

The algorithm will continue to explore all possible placements until it finds a configuration where all eight queens are placed without conflicts. One possible final arrangement could look like this:



(OR)

(b) Explain A* algorithm and find the shortest path from ‘a to z ‘ using the following graph.

Components of A* Algorithm

Graph Representation: The algorithm operates on a graph, which can be represented as nodes (or vertices) connected by edges.

Cost Functions:

g(n): The actual cost from the start node to node

n

n.

h(n): A heuristic estimated cost from node

n

n to the target node. This must be an admissible heuristic, meaning it never overestimates the true cost.

f(n): The total estimated cost of the cheapest solution through node

n

n:

f

(

n

)

=

g

(

n

)

+

h

(

n

)

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

Open List: A priority queue that contains nodes that need to be evaluated, sorted by their

f

(

n

)

f(n) values.

Closed List: A list of nodes that have already been evaluated.

Steps of the A* Algorithm

Initialization: Start with the open list containing the start node. Set

g

(

s

t

a

r

t

)

=

0

g(start)=0 and calculate

f

(

s

t

a

r

t

)

=

h

(

s

t

a

r

t

)

f(start)=h(start).

Main Loop:

While the open list is not empty:

Select Node: Remove the node

n

n with the lowest

f

(

n

)

f(n) from the open list.

Goal Check: If

n

n is the target node, reconstruct the path and return it.

Generate Successors: For each neighbor of

n

n:

Calculate

g

g score (actual cost) to reach the neighbor.

If the neighbor is not in the open list or has a lower

g

g score, update its

g

g,

h

h, and

f

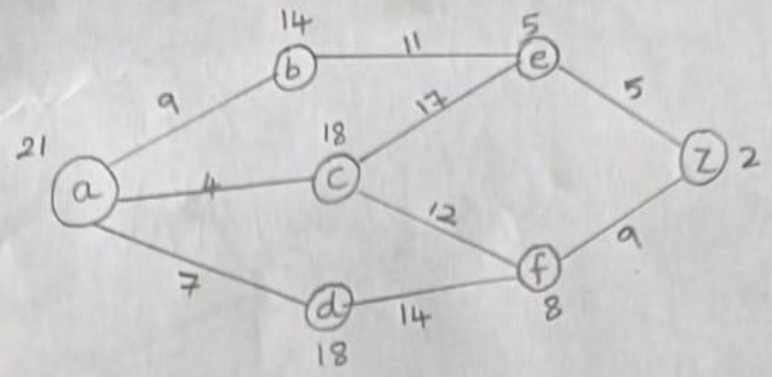
f values, and add it to the open list if not already present.

Move

n

n to the closed list.

Path Reconstruction: Once the target is found, backtrack from the target node to the start node using parent pointers to reconstruct the path.



a	21
b	14
c	18
d	18
e	5
f	8
z	2

Iteration : 1

$a-b = 9+14 = 23$

$a-c = 4+18 = 22$

$a-d = 7+18 = 25$

Iteration : 2

$a-c-e = 21+5 = 26$

$a-c-f = 16+8 = 24$

Iteration : 3

$a-c-f-z = 25+2 = 27$