Artificial Intelligence

CSE 3317

Avisheak Das

Ref: *Artificial Intelligence - A Modern Approach*

Lecture 1:

Definition of AI

Artificial Intelligence (AI) is the science and engineering of creating intelligent machines that can perform tasks that typically require human intelligence, such as reasoning, learning, problem-solving, and understanding language.

AI Subfields

AI encompasses various specialized subfields:

1. **Machine Learning (ML):**
   Focuses on algorithms that allow computers to learn patterns and make decisions from data without explicit programming.
2. **Deep Learning:**
   A subset of ML involving neural networks with many layers to process complex data like images, speech, or text.
3. **Natural Language Processing (NLP):**
   Enables machines to understand and interact with human language, such as translation or sentiment analysis.
4. **Robotics:**
   Involves designing intelligent machines that can interact with the physical world (e.g., robots navigating an environment).
5. **Neural Networks and Fuzzy Logic:**
   - **Neural Networks:** Model the brain's structure to enable learning and decision-making.
   - **Fuzzy Logic:** A method to handle uncertain or imprecise information in decision-making.
6. **Expert Systems:**
   AI systems designed to emulate decision-making by human experts in specific domains (e.g., medical diagnosis).

Key Concepts

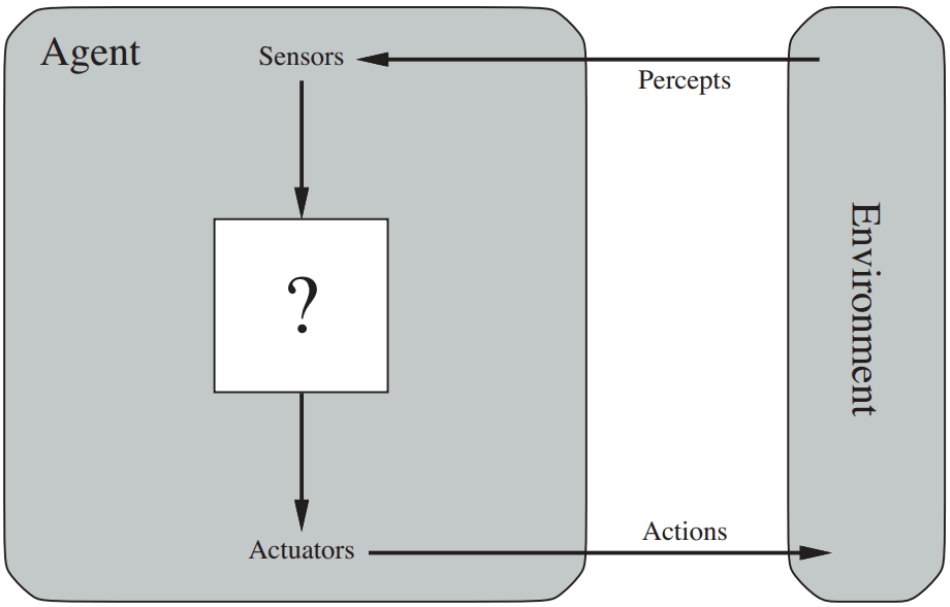
- **Agent:**
  An entity that perceives its environment through sensors and acts upon it using actuators.
- **Environment:**
  The external context in which an agent operates.

Lecture 2:

Agent-Environment Interaction

- Agents operate in an environment by perceiving it and acting to achieve specific goals.

Agent and Environment Diagram



- Shows how an agent interacts with the environment through a loop of:
    1. **Perception:** Observing the environment.
    2. **Action:** Acting upon the environment.
    3. **Learning:** Updating its knowledge or behavior.

PEAS Framework

PEAS defines the task environment for an intelligent agent. It specifies four components to design and analyze intelligent systems:

1. Performance Measure

- Criteria used to evaluate the agent's success.

2. Environment

- The context in which the agent operates.

3. Actuators

- Tools or mechanisms the agent uses to act on the environment.

4. Sensors

- Devices or methods the agent uses to perceive the environment.

PEAS Examples

Technology	Performance Measure	Environment	Actuators	Sensors
Autonomous Car	Safety, time, fuel efficiency, comfort	Roads, traffic, pedestrians	Steering, accelerator, brakes	Cameras, radar, GPS, lidar
Vacuum Cleaning Robot	Cleanliness, battery usage	Floors, furniture, walls	Wheels, suction motor, brushes	Cameras, bump sensors, dirt sensor
Spam Email Filter	High accuracy in detecting spam	Email database	Classifier marking emails as spam	Text content, metadata
Chess AI	Winning games, strategy improvement	Chessboard, opponent moves	Move chess pieces	Board state, opponent's moves
Personal Assistant (e.g., Siri)	Correct responses, user satisfaction	User input (voice/text), internet	Voice/text output	Microphone, text parser

---

Types of Environments

1. **Observability:**
    - **Fully Observable:** Complete state of the environment is available (e.g., chessboard).
    - **Partially Observable:** Only part of the environment is visible (e.g., self-driving car with limited sensors).
2. **Agent Interaction:**
    - **Single-Agent:** One agent acts in the environment (e.g., vacuum cleaner).
    - **Multi-Agent:** Multiple agents interact, either cooperatively or competitively (e.g., online multiplayer games).

3. **Determinism:**
    - **Deterministic:** Actions have predictable outcomes (e.g., playing Sudoku).
    - **Stochastic:** Outcomes are uncertain or probabilistic (e.g., stock trading agent).
4. **Temporal Nature:**
    - **Episodic:** Each action is independent (e.g., image classification).
    - **Sequential:** Actions are interconnected and affect future actions (e.g., route planning for a delivery drone).
5. **Environment Dynamics:**
    - **Static:** Environment does not change over time (e.g., crossword puzzle).
    - **Semi-Dynamic:** Environment changes, but not due to the agent's actions (e.g., a turn-based game).
    - **Dynamic:** Environment evolves continuously and unpredictably (e.g., real-time traffic navigation).
6. **Discreteness:**
    - **Discrete:** Environment has a finite number of states or actions (e.g., board games).
    - **Continuous:** Environment has infinite possibilities (e.g., robot navigation in a real-world space).

Lecture 3

Uninformed Search Techniques

Uninformed search algorithms explore the search space without any domain-specific knowledge (heuristics). They only use the information provided by the problem definition.

---

1. Breadth-First Search (BFS)

Description:

- Explores all nodes at the current depth before moving to the next depth level.
- Uses a **queue** data structure (FIFO).

When to Use:

- When the shortest path in an **unweighted graph** is required.
- When all edges have the same cost.

Pseudo Code:

```
function BFS(graph, start, goal):
    queue ← [start]   // Initialize queue with the start node
    visited ← {}      // Keep track of visited nodes

    while queue is not empty:
        current ← queue.pop(0)     // Dequeue the first node

        if current == goal:
            return "Path Found!"   // Goal reached

        for each neighbor in graph[current]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

    return "Path Not Found"
```

Complexity:

- Time: $O(b^d)$, where:
  - b: Branching factor (average number of children per node).
  - d: Depth of the shallowest solution.
- Space: $O(b^d)$ (stores all nodes in memory).

---

2. Depth-First Search (DFS)

Description:

- Explores a path as deeply as possible before backtracking.
- Uses a **stack** data structure (can use recursion).

When to Use:

- When the search space is large, and a **solution exists deep in the tree**.
- Space optimization is required.

Pseudo Code:

plaintext
Copy code

```
- function DFS(graph, start, goal):
-     stack ← [start]    // Initialize stack with the start node
-     visited ← {}       // Keep track of visited nodes
-
-     while stack is not empty:
-         current ← stack.pop()      // Pop the last node
-
-         if current == goal:
-             return "Path Found!"   // Goal reached
-
-         for each neighbor in graph[current]:
-             if neighbor not in visited:
-                 visited.add(neighbor)
-                 stack.append(neighbor)
-
-     return "Path Not Found"
```

Complexity:

- Time: O(bm)O(b^m)O(bm), where mmm is the maximum depth of the tree.
- Space: O(b×m)O(b \times m)O(b×m) (only needs to store the current path).

Quirks:

- May not find the shortest path.
- Risk of getting stuck in infinite loops (use limited-depth DFS or iterative deepening).

Informed Search Techniques

Informed search algorithms use **heuristic functions** to estimate the cost of reaching the goal, guiding the search toward optimal solutions.

---

3. Uniform Cost Search (UCS)

Description:

- Explores nodes based on their cumulative cost from the start node.
- Expands the least-cost node first using a **priority queue**.

When to Use:

- When edge costs are not uniform, and the optimal solution is required.

Pseudo Code:

```
- function UCS(graph, start, goal):
-     priorityQueue ← [(0, start)]  // Queue stores (cost, node)
-     visited ← {}
-
-     while priorityQueue is not empty:
-         (cost, current) ← priorityQueue.pop() // Pop node with
  least cost
-
-         if current == goal:
-             return "Path Found with Cost: " + cost
-
-         for each neighbor in graph[current]:
-             newCost ← cost + graph[current][neighbor]  // Edge
  cost
-             if neighbor not in visited or newCost <
  visited[neighbor]:
-                 visited[neighbor] = newCost
-                 priorityQueue.push((newCost, neighbor))
-
-     return "Path Not Found"
```

Complexity:

- Time: $O(b^d)$.
- Space: $O(b^d)$.

Quirks:

- Guarantees optimal solutions if all edge costs are non-negative.

---

4. Best-First Search (BeFS)

Description:

- Selects nodes based on their heuristic value (h(n)h(n)h(n)), where h(n)h(n)h(n) estimates the cost to reach the goal.

When to Use:

- When heuristic values can provide meaningful guidance to the goal.

Pseudo Code:

plaintext
Copy code

```
- function BeFS(graph, start, goal, heuristic):
-     priorityQueue ← [(heuristic(start), start)]  // Queue
  stores (h, node)
-
-     while priorityQueue is not empty:
-         (hValue, current) ← priorityQueue.pop()  // Pop node
  with lowest h
-
-         if current == goal:
-             return "Path Found!"
-
-         for each neighbor in graph[current]:
-             hNeighbor ← heuristic(neighbor)
-             priorityQueue.push((hNeighbor, neighbor))
-
-     return "Path Not Found"
```

Complexity:

- Time: $O(b^m)$, where m is the maximum depth.
- Space: $O(b^m)$.

Quirks:

- May not guarantee the shortest path unless combined with cumulative costs *f(n) = g(n) + h(n)* .

---

*5. A Search**

Description:

- Combines the strengths of UCS and BeFS using *f(n)=g(n)+h(n)*, where:
  - g(n): Cost from the start node to n.
  - h(n): Heuristic estimate of the cost from n to the goal.

When to Use:

- When both actual costs and heuristic estimates are meaningful.
- Guarantees optimal solutions when h(n) is **admissible** and **consistent**.

Pseudo Code:

```
- function AStar(graph, start, goal, heuristic):
-     priorityQueue ← [(heuristic(start), 0, start)]  // Queue
  stores (f, g, node)
-     visited ← {}
-
-     while priorityQueue is not empty:
-         (fValue, gValue, current) ← priorityQueue.pop()  //
  Pop node with lowest f
-
-         if current == goal:
-             return "Path Found with Cost: " + gValue
-
-         for each neighbor in graph[current]:
-             gNeighbor ← gValue + graph[current][neighbor]  //
  Actual cost
-             hNeighbor ← heuristic(neighbor)
-             fNeighbor ← gNeighbor + hNeighbor
-             if neighbor not in visited or gNeighbor <
  visited[neighbor]:
-                 visited[neighbor] = gNeighbor
-                 priorityQueue.push((fNeighbor, gNeighbor,
  neighbor))
-
-     return "Path Not Found"
```

Complexity:

- Time: $O(b^d)$.
- Space: $O(b^d)$.

Quirks:

- Guarantees optimal solutions but can be memory-intensive.

Deriving $O(b^d)$ Complexity

1. **Tree Representation**:
    - A search tree has b branches at each level.
    - At depth d, there are b^d nodes.
2. **Total Nodes Explored**:
    - BFS and UCS explore all nodes up to depth d : $b^0+b^1+b^2+\cdots+b^d=O(b^d)1 + b + b^2 +\cdots+ b^d = O(b^d)$
3. **Time Complexity**:
    - Each node requires constant-time operations (enqueue, dequeue, expand).
    - Hence, the overall complexity is proportional to $O(b^d)$.
4. **Space Complexity**:
    - BFS and UCS store all frontier nodes, leading to space usage of $O(b^d)$.

Lecture 4

Breadth-First Search (BFS)

BFS explores all neighbors of a node before moving to their successors, ensuring a level-order traversal.

Pseudocode:

```
BFS(Graph, Start_Node):
    Create a queue (Q) and add Start_Node
    Mark Start_Node as visited
    While Q is not empty:
        Current_Node = Dequeue(Q)
        Process Current_Node
        For each neighbor of Current_Node:
            If neighbor is not visited:
          Mark neighbor as visited
                Enqueue(Q, neighbor)
```

Graph:

```
A -> {B, C}
B -> {D, E}
C -> {F, G}
```

Start Node: `A`

Step-by-step Execution:

1. Initialize the queue: `Q = [A]`, mark `A` as visited.
2. Dequeue `A`, process it: `Visited = {A}`. Enqueue neighbors: `Q = [B, C]`.
3. Dequeue `B`, process it: `Visited = {A, B}`. Enqueue neighbors: `Q = [C, D, E]`.
4. Dequeue `C`, process it: `Visited = {A, B, C}`. Enqueue neighbors: `Q = [D, E, F, G]`.
5. Dequeue `D`, process it: `Visited = {A, B, C, D}`. (No new neighbors.)
6. Dequeue `E`, process it: `Visited = {A, B, C, D, E}`. (No new neighbors.)
7. Dequeue `F`, process it: `Visited = {A, B, C, D, E, F}`. (No new neighbors.)
8. Dequeue `G`, process it: `Visited = {A, B, C, D, E, F, G}`.

Order of Nodes Visited: A → B → C → D → E → F → G.

Logical Search (with Greedy Approach)

Logical Search:

- Logical search uses inference and reasoning to derive conclusions or explore states.
- **Greedy Search**: Focuses on optimizing a heuristic at each step, aiming to find the local optimum.

Greedy Approach Example:

Scenario: Navigate a graph to find the shortest path from **Start (S)** to **Goal (G)**.
Graph:

```
S -> A (Cost: 1)
A -> G (Cost: 2)
S -> B (Cost: 5)
B -> G (Cost: 1)
```

Heuristic: Distance from each node to `G`:

$$\mathrm{h(S)} = 3, \mathrm{h(A)} = 2, \mathrm{h(B)} = 1, \mathrm{h(G)} = 0$$

Execution:

1. Start at `S`. Compare `A` ($h(A) = 2$) and `B` ($h(B) = 1$), choose `B`.
2. At `B`, move to `G` as it has the least $h(G) = 0$.

Path: $S \rightarrow B \rightarrow G$.

Constraint Satisfaction Problem (CSP)

A CSP involves variables, domains, and constraints:

- **Variables**: X1,X2,...,Xn
- **Domains**: Possible values for each variable.
- **Constraints**: Conditions that limit variable assignments.

Forward Checking in CSP:

- During variable assignment, eliminate inconsistent values from the domains of unassigned variables.

Example: 3-Coloring Problem

Scenario: A graph with 3 nodes $\{A, B, C\}$, and the edges $(A, B), (B, C), (C, A)$.
Colors: $\{Red, Green, Blue\}$.

Step-by-Step Forward Checking:

1. Assign $A = Red$. Remove `Red` from B's and C's domains.

$$B \in \{Green, Blue\},\ C \in \{Green, Blue\}$$

2. Assign $B = Green$. Remove `Green` from C's domain.

$$C \in \{Blue\}$$

3. Assign $C = Blue$.

Basics of N-Queens Problem

Objective: Place N queens on an N×N chessboard such that no two queens attack each other.

Rules:

- Only one queen per row, column, or diagonal.

Example: 4-Queens Problem

Chessboard: 4×4.

Approach:

1. Place the first queen at (1,1).
2. Place the second queen in the second row, avoiding the same column or diagonal as the first.
3. Continue until all queens are placed.
4. Backtrack if no valid position exists.

Pseudocode for N-Queens:

```
SolveNQueens(board, row):
    If all rows are filled:
        Print solution
    For each column in the row:
        If placing queen at (row, column) is safe:
            Place queen
            Recur for the next row
            If solution found, return
            Remove queen (Backtrack)
```

Key Concepts:

1. **Backtracking**: Systematically explore options and backtrack when constraints are violated.
2. **Forward Checking**: Eliminate invalid positions based on previously placed queens.

Solution for 4-Queens:

```
Solution: Q at positions (1,2), (2,4), (3,1), (4,3).
```