AIES unit 2 imp stuff

Knowledge Representation (KR) involves encoding information about the world in a form that an AI system can understand, reason with, and use to solve problems. It allows AI agents to store, retrieve, and manipulate knowledge to make decisions. It bridges the gap between raw data and actionable intelligence by structuring information in a logical and computationally efficient manner.

A Knowledge Base (KB) stores these representations as sentences in a knowledge representation language, which has syntax (rules for forming valid sentences) and semantics (meaning of sentences in the context of the world).

KR involves:

- **Syntax:** The structure or format of the representation.
- **Semantics:** The meaning of the symbols and structures.
- **Inference:** The ability to deduce new knowledge from existing knowledge.
- **Pragmatics:** How knowledge is applied to solve real-world problems.

Intelligent agents should have capacity for:

- **Perceiving**, that is, acquiring information from environment
- **Knowledge Representation**, that is, representing its understanding of the world
- **Reasoning**, that is, inferring the implications of what it knows and of the choices it has
- **Acting**, that is, choosing what it want to do and carry it out

Logics are formal languages for representing information such that conclusions can be drawn.

- **Syntax** defines the sentences in the language.
- **Semantics** define the "meaning" of sentences

Types of Logic:

1. **Propositional Logic**
    a. It is a symbolic logic dealing with propositions. It is concerned with declarative sentences that are either true or false.

    b. It uses propositional symbols (e.g., P, Q) and logical connectives (e.g., $\wedge$, $\vee$, $\neg$, $\rightarrow$) to form compound statements

    c. **AND ($\wedge$)**: P $\wedge$ Q (Both P and Q must be true)
    OR ($\vee$): P $\vee$ Q (At least one of P or Q must be true)
    NOT ($\neg$): $\neg$P (P is false)
    IMPLIES ($\rightarrow$): P $\rightarrow$ Q (If P is true, then Q must be true)
    EQUIVALENCE ($\equiv$): P $\equiv$ Q (P and Q have the same truth value)

d. **Example**:
    i. P: “It is hot”, Q: “It is humid”, R: “It is raining”.
    ii. Sentence: P ∧ Q → R (“If it is hot and humid, then it is raining”).
    iii. Truth Table: Evaluates P → Q (e.g., true when P is false or Q is true).
e. Propositional logic is declarative
   Propositional logic allows partial/disjunctive/negated information
   Propositional logic is compositional
   Propositional logic has very limited expressive power

2. **Predicate Logic**
    a. Predicate Logic extends Propositional Logic by including objects, properties, and relations. It allows for more expressive representations using quantifiers:
        i. **Universal Quantifier (∀)** — Represents "for all" statements.
        ii. **Existential Quantifier (∃)** — Represents "there exists" statements.
    b. Example:
        i. "All humans are mortal" is represented as: **∀x(Human(x) -> Mortal(x))**
        ii. "Some humans are doctors" is represented as: **∃x(Human(x) ^Doctor(x))**
    c. Predicate Logic is more expressive than Propositional Logic as it captures relationships and properties of objects, not just statements of truth.

Inference Engine
It is a component of the system that applies logical rules to the knowledge base to deduce new information.

1. **Forward Chaining**
    a. A data-driven inference method that **starts with known facts** and applies rules to derive new facts until the goal is reached
    b. **Process**:
        i. Start with KB facts.
        ii. Match facts to rule premises.
        iii. Apply rules to add facts.
        iv. Repeat until goal or no new facts
2. **Backward Chaining**
    a. Backward chaining is a goal-driven inference method. It starts with the goal and works backward to determine if existing facts and rules can prove it.
    b. **Process**:
        i. Start with goal.
        ii. Find rules matching goal.
        iii. Prove premises as subgoals.
        iv. Repeat until facts reached.

- ## Example: Animal Classification System

Knowledge Base (KB):

Facts:

- Fact 1: Has_Feathers(Bird) (The animal has feathers).
- Fact 2: Lays_Eggs(Bird) (The animal lays eggs).

Rules:

- Rule 1: Has_Feathers(X) $\wedge$ Lays_Eggs(X) $\rightarrow$ Is_Bird(X) (If an animal has feathers and lays eggs, it is a bird).
- Rule 2: Is_Bird(X) $\rightarrow$ Can_Fly(X) (If an animal is a bird, it can fly).

Goal: Determine if Can_Fly(Bird) is true (i.e., can the animal fly?).

Forward Chaining Example

Question: Use forward chaining to prove Can_Fly(Bird).

Steps:

1. **Initial Facts**: Has_Feathers(Bird), Lays_Eggs(Bird).
2. **Check Rules**:
   - Rule 1: Has_Feathers(Bird) $\wedge$ Lays_Eggs(Bird) is true (both facts exist).
   - Apply Rule 1: Add new fact Is_Bird(Bird) to the KB.
3. **Check Rules Again**:
   - Rule 2: Is_Bird(Bird) is true (from the new fact).
   - Apply Rule 2: Add new fact Can_Fly(Bird) to the KB.
4. **Goal Check**: Can_Fly(Bird) is now in the KB, so the goal is proven.
5. **Stop**: No further rules to apply.

Result: Can_Fly(Bird) is true (the animal can fly).

Explanation:

- Forward chaining starts with the facts (feathers, eggs) and systematically applies rules to derive Is_Bird(Bird), then Can_Fly(Bird). It's like a doctor collecting symptoms and building a diagnosis step-by-step.

Backward Chaining Example

Question: Use backward chaining to prove Can_Fly(Bird).

Steps:

1. **Goal**: Can_Fly(Bird).
2. **Find Rule for Goal**:
    - Rule 2: Is_Bird(X) → Can_Fly(X) can prove Can_Fly(Bird) if Is_Bird(Bird) is true.
    - New subgoal: Prove Is_Bird(Bird).
3. **Find Rule for Subgoal**:
    - Rule 1: Has_Feathers(X) ∧ Lays_Eggs(X) → Is_Bird(X) can prove Is_Bird(Bird) if Has_Feathers(Bird) and Lays_Eggs(Bird) are true.
    - New subgoals: Prove Has_Feathers(Bird) and Lays_Eggs(Bird).
4. **Check Subgoals**:
    - Has_Feathers(Bird) is a fact in the KB (true).
    - Lays_Eggs(Bird) is a fact in the KB (true).
5. **Backtrack**:
    - Rule 1 premises are true, so Is_Bird(Bird) is true.
    - Rule 2 premise (Is_Bird(Bird)) is true, so Can_Fly(Bird) is true.
6. **Stop**: Goal Can_Fly(Bird) is proven.

Result: Can_Fly(Bird) is true (the animal can fly).

Explanation:

- Backward chaining starts with the goal (can it fly?) and works backward, checking if the animal is a bird, then verifying the conditions for being a bird (feathers, eggs). It's like answering a question by tracing back to known facts, as in a query system like Prolog.

Logic Programming

Logic Programming is a programming paradigm based on formal logic. Programs are written as a set of logical statements, and solutions are derived by logical inference.
It uses languages like Prolog.

Key Features:

- **Facts**: Simple assertions (e.g., father(john, mary)).
- **Rules**: Logical implications (e.g., parent(X, Y) :- father(X, Y)).

- **Queries**: Questions to the system (e.g., ?- parent(john, mary)).
- **Inference**: Uses backward chaining to resolve queries by unifying facts and rules (Page 9).
- **Syntax**: Horn clauses (e.g., A :- B1, B2, …, Bn, meaning B1 ∧ B2 ∧ … ∧ Bn → A).
- **Example:** criminal(X) :- american(X), weapon(Y), sells(X, Y, hostile(Z)).

Resolution in Predicate Logic

Resolution in Predicate Logic is a method of logical inference used to prove that a given statement is true by showing that its negation leads to a contradiction. The goal is to prove that a set of premises leads to a contradiction.

Process:

1. Convert statements to clause form.
2. Negate the goal and add it to the knowledge base.
3. Apply unification and resolve clauses until an empty clause (contradiction) is found.

Unification

Unification is the process of finding a substitution that makes two logical expressions identical. It is used in automated reasoning and logic programming to make predicates match.

Process:

1. Compare two expressions (e.g., predicates P(x, f(y)), P(a, f(b))).
2. If predicates differ, unification fails.
3. If arguments differ:
    - Variable and constant: Substitute variable with constant (e.g., x → a).
    - Two variables: Substitute one with the other (e.g., x → y).
    - Function terms: Recursively unify arguments (e.g., f(y) and f(b) → y → b).
4. Apply substitutions consistently across all terms.
5. Output the **Most General Unifier (MGU)**, the substitution with minimal constraints.

Example:

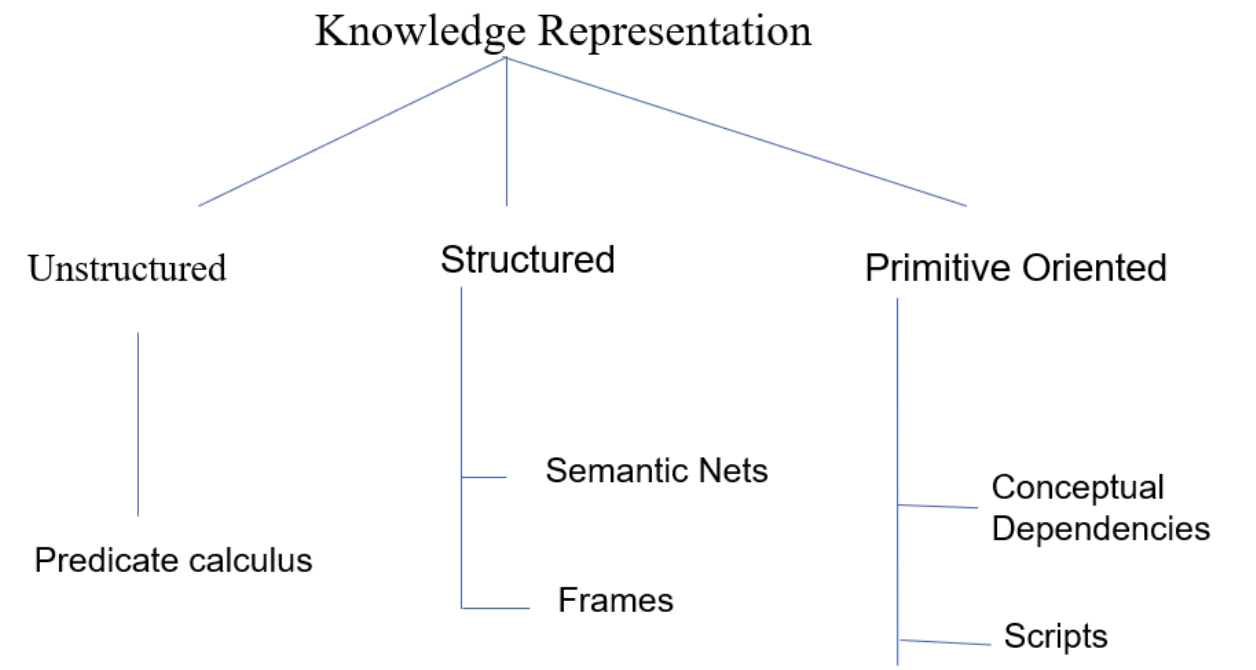
For predicates:

- loves(John,x)
- loves(John,Mary)

Unification will match x=Mary, making both expressions compatible.

Knowledge Representation

Classification



Knowledge Representation (KR) is the process of encoding information about the world in a computer-tractable form to enable reasoning and decision-making. KR structures organize knowledge to support inference, querying, and problem-solving in AI systems.

Predicate Calculus

Predicate Calculus (First-order logic) is a powerful method of knowledge representation that allows for the expression of relationships between objects, properties, and their attributes. It extends propositional logic by using predicates, variables, and quantifiers.

Components of Predicate Calculus:

1. **Predicates:** Describe properties of objects or relationships between objects (e.g., `Human(Socrates)`).
2. **Variables:** Represent objects (e.g., x, y).
3. **Quantifiers:**

- Universal Quantifier (∀): States that a property is true for all elements (e.g., ∀x Human(x)).
- Existential Quantifier (∃): States that a property is true for at least one element (e.g., ∃x Human(x)).

Example:

"All humans are mortal" can be represented as: ∀x(Human(x) -> Mortal(x))

"Some people are doctors" can be represented as: ∃x(Human(x) ^ Doctors(x))

Semantic Networks

Semantic Networks are graphical representations of knowledge that illustrate relationships between different concepts. Concepts are represented as **nodes**, and relationships are represented as **edges**.

Characteristics:

1. **Inheritance:** If a concept is connected to another through an "is-a" relationship, it inherits its properties. For example, if `Bird` is a `Animal`, all birds inherit the characteristics of animals.
2. **Link Types:** Different types of links represent different relationships like `is-a`, `a-kind-of`, `part-of` and `an-instance-of`.

Example:

A semantic network for birds might represent:

- `Sparrow is-a Bird`
- `Bird is-a Animal`
- `Bird can Fly`

This structure allows for quick navigation through relationships.

Frames

Frames are a collection of attributes and associated values that describe some entity in the world. A frame may involve procedural embedding in place of values of attributes. Frames are linked by relations of specialization/generalization and by many ad-hoc relations

Characteristics:

1. **Attributes (Slots):** These are properties or features of an object.
2. **Values (Fillers):** The specific values associated with each slot.
3. **Inheritance:** Frames can inherit attributes from other frames, making knowledge reuse easier.

Example:

A "Car" frame might have the following slots:

- **Color:** Red
- **Model:** Tesla Model 3
- **Owner:** John Doe

Frames can be linked to each other to form a hierarchical structure.

Conceptual Dependency (CD)

Conceptual Dependency is a theory proposed by Roger Schank for representing knowledge and understanding natural language through primitive actions and conceptualizations. CD helps to capture the meaning of sentences independent of the specific words used.

Primitive Acts in CD:

●ATRANS	Transfer of an abstract relationship (i.e. give)
●PTRANS	Transfer of the physical location of an object (e.g., go)
●PROPEL	Application of physical force to an object (e.g. push)
●MOVE	Movement of a body part by its owner (e.g. kick)
●GRASP	Grasping of an object by an action (e.g. throw)
●INGEST	Ingesting of an object by an animal (e.g. eat)
●EXPEL	Expulsion of something from the body of an animal (e.g. cry)
●MTRANS	Transfer of mental information (e.g. tell)
●MBUILD	Building new information out of old (e.g decide)
●SPEAK	Producing of sounds (e.g. say)
●ATTEND	Focusing of a sense organ toward a stimulus (e.g. listen)

Example:

"John gave a book to Mary" is represented as:

- Action: ATRANS
- Object: Book
- Agent: John
- Recipient: Mary

Conceptual Categories (CC)

Conceptual Categories are part of the Conceptual Dependency (CD) theory that helps in structuring and classifying actions and objects. It consists of four main categories:

1. **ACT (Actions):** Represents physical or mental actions (e.g., GIVE, MOVE).
2. **PP (Picture Producers):** Represents objects involved in actions (e.g., a person, a book).
3. **AA (Action Aiders):** Represents attributes of actions (e.g., speed, intensity).
4. **PA (Picture Aiders):** Represents properties or attributes of objects (e.g., color, size).

Example:

In the sentence "John gave a book to Mary," the conceptual categories would be:

- ACT: ATRANS (give)
- PP: John (agent), Mary (recipient), Book (object)
- AA: Method of giving (e.g., handover)

These categories help in making complex event representations more structured and easily understandable.

Conceptual Graphs (CG)

Conceptual Graphs are a formalism for representing knowledge in a network form. They are used to express relationships and attributes in a visual way that is easy to interpret.

Components of CG:

1. **Concept Nodes:** Represent objects or ideas.
2. **Relation Nodes:** Represent relationships between concepts.

Example:

To represent "The cat is on the mat":

- Concept Nodes: `Cat`, `Mat`
- Relation Node: `On`

Scripts

Scripts are structured representations of stereotypical sequences of events in specific contexts. They outline typical procedures in everyday activities, allowing AI systems to predict and understand sequences of events.

Example:

A `Restaurant Script` might include:

1. Entering the restaurant
2. Being seated
3. Ordering food
4. Eating
5. Paying the bill
6. Leaving the restaurant

Scripts allow AI to fill in gaps in understanding by predicting the next logical step in the sequence.

Planning

Planning in AI is about the decision making tasks performed by the robots or computer programs to achieve a specific goal.
The execution of planning is about choosing a sequence of actions with a high likelihood of completion of the specific task.

Key Elements of Planning:

- **Initial State**: The starting configuration of the environment.
- **Goal State**: The desired configuration or outcome.
- **Actions/Operators**: Defined by their preconditions (what must be true to execute the action) and effects (how the action changes the state).
- **Plan**: An ordered list of actions that transforms the initial state into the goal state.

Example:
Imagine a robot that needs to move an object from Location A to Location B. The robot must:

1. Pick up the object (Action 1)
2. Navigate through obstacles (Action 2)
3. Place the object at Location B (Action 3)

Forward Planning

Forward Planning (State Space Forward Search) starts with the **initial state** and applies actions step by step to reach the **goal state**. The planner explores all possible actions from the current state and continues until the goal is achieved.

Process:

1. Begin at the initial state.
2. Apply all possible actions that are valid in that state.
3. Transition to the next state.
4. Repeat until the goal state is reached or no more actions are possible.

Example:

A robot in a grid world needs to reach a target cell. Starting from (0,0), it tries actions like 'Move Right', 'Move Down', until it reaches the goal cell (2,2).

Backward Planning

Backward Planning (State Space Backward Search) starts with the **goal state** and works backward to identify the necessary conditions that must be true in the preceding steps. It continues to backtrack until it reaches the initial state.

Process:

1. Begin with the goal state.
2. Identify the required preconditions for the goal.
3. Apply actions that can achieve those preconditions.
4. Repeat until the initial state is found.

Example:

If the goal is to be at Location B with an object, Backward Planning starts with this requirement and finds that the robot needs to first be at Location B and hold the object. This continues backward until the initial state is mapped out.

Goal Stack Planning

Goal Stack Planning uses a **stack-based structure** to keep track of goals and sub-goals. Goals are pushed onto the stack, and the planner pops each goal, attempts to satisfy it, and pushes any sub-goals required for its fulfillment back onto the stack.

Process:

1. Push the main goal onto the stack.
2. If the goal is complex, push its sub-goals onto the stack.
3. Solve each sub-goal, possibly introducing new sub-goals.
4. Continue until all sub-goals are satisfied and the main goal is achieved.

Example:

To "Make a Cup of Coffee":

- Main goal: `Make Coffee`
- Sub-goals: `Boil Water`, `Add Coffee`, `Pour Water`, `Stir`
- Each sub-goal is popped from the stack and executed in order.

Hierarchical Planning

Hierarchical Planning organizes actions into a **hierarchy**, decomposing complex, high-level actions into simpler, low-level (executable) actions using predefined schema.
It reduces complexity by planning at abstract levels before refining details.

Process:

1. Start with a high-level goal.
2. Decompose the goal into smaller sub-tasks.
3. Solve each sub-task individually.
4. Combine sub-task solutions to achieve the main goal.

Example:

To plan a "Travel Trip":

- Main goal: `Travel to France`
- Sub-goals: `Book Flight`, `Pack Bags`, `Arrange Transport`
- Each of these sub-goals can further be broken down, like `Book Flight` → `Search Flights`, `Select Flight`, `Confirm Booking`.

Criteria	Forward Planning	Backward Planning
Starting Point	Begins from the **initial state** and progresses forward step by step towards the goal.	Begins from the **goal state** and traces backward to identify necessary initial conditions.
Search Direction	Moves forward through state transitions by applying actions.	Moves backward through actions that could lead to the goal state.
Efficiency	Efficient when there are fewer paths to explore from the initial state.	Efficient when there are fewer sub-goals and the goal is well-defined.
Goal Focused?	Not necessarily goal-focused; explores all possibilities from the start.	Highly goal-focused; only traces paths that are directly connected to achieving the goal.
Memory Usage	Typically uses more memory as it explores many forward states.	Generally uses less memory by focusing on paths relevant to the goal.
Example	A robot navigating a maze from the entrance to the exit step by step.	A chess player thinking backward from a checkmate position to find the right moves.
Best Use Cases	When the initial state is well-defined and straightforward.	When the goal state is well-defined and sub-goals are clearly identifiable.

Criteria	Propositional Logic	Predicate Logic
Definition	A logic that deals with propositions that are either true or false.	An extension of propositional logic that includes objects, relations, and quantifiers.
Representation of Facts	Simple statements or propositions.	Uses predicates to represent relationships and properties of objects.
Symbols Used	Logical connectives like ∧ (AND), ∨ (OR), ¬ (NOT), → (IMPLIES), ↔ (IFF).	Uses predicates (e.g., `Human(Socrates)`), variables (e.g., `x`, `y`), and quantifiers (∀, ∃).
Expressiveness	Limited to true/false statements; cannot represent complex relationships.	More expressive; can represent complex statements like "All humans are mortal".
Quantifiers	Not supported.	Supports **Universal (∀)** and **Existential (∃)** quantifiers.
Example	P → Q (If it rains, the ground gets wet).	∀x (Human(x) → Mortal(x)) — "All humans are mortal".
Domain of Application	Used for simple logical statements and truth analysis.	Suitable for knowledge representation in AI, databases, and expert systems.
Limitations	Cannot express relationships between objects.	Can express relationships, attributes, and more complex knowledge structures.