**EX.NO 4 a) IMPLEMENT FORWARD CHAINING STRATEGIES**

**INTRODUCTION:**

An expert system is a computer application that uses rules, approaches and facts to provide solutions to complex problems. An expert system contains two primary components:

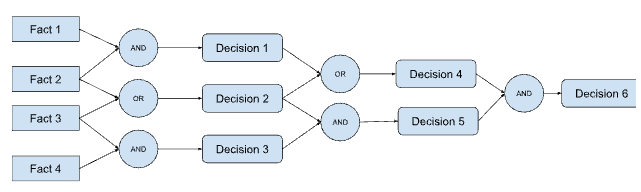
1. **Knowledge Base:** This is a structured collection of facts about the system’s domain.
2. **Inference engine:** This is a component of the expert system that applies logical rules to the knowledge base to deduce new information. It interprets and evaluates the facts in the knowledge base in order to provide an answer

Forward chaining and backward chaining are two strategies used in designing expert systems for AI.

**DEFINITION:**

Forward chaining is a form of reasoning that starts with simple facts in the knowledge base and applies inference rules in the forward direction to extract more data until a goal is reached.

* Forward chaining is also known as a **forward deduction or forward reasoning method** when using an inference engine. The forward-chaining algorithm starts from known facts, triggers all rules whose premises are satisfied and adds their conclusion to the known facts. This process repeats until the problem is solved.
* In this type of chaining, the inference engine starts by evaluating existing facts, derivations, and conditions before deducing new information. An endpoint, or goal, is achieved through the manipulation of knowledge that exists in the knowledge base.



**PROPERTIES:**

* Forward chaining follows a down-up strategy, from bottom to top.
* It uses known facts to start from the initial state (facts) and works toward the goal state, or conclusion.
* The forward chaining method is also known as data-driven because achieve our objective by employing available data.
* The forward chaining method is widely used in expert systems such as CLIPS, business rule systems and manufacturing rule systems.
* It uses a breadth-first search as it has to go through all the facts first.
* It can be used to draw multiple conclusions.

**PROGRAM:**

**class Rule:**

**def \_\_init\_\_(self, premises, conclusion):**

**self.premises = premises**

**self.conclusion = conclusion**

**def evaluate(self, facts):**

**for premise in self.premises:**

**if premise not in facts:**

**return False**

**return True**

**def fire(self, facts):**

**facts.add(self.conclusion)**

**class ForwardChaining:**

**def \_\_init\_\_(self, rules):**

**self.rules = rules**

**self.facts = set()**

**def run(self):**

**while True:**

**fired = False**

**for rule in self.rules:**

**if rule.evaluate(self.facts):**

**rule.fire(self.facts)**

**fired = True**

**if not fired:**

**break**

**def main():**

**rules = [**

**Rule(["mammal(A)"], "vertebrate(A)"),**

**Rule(["vertebrate(A)"], "animal(A)"),**

**Rule(["vertebrate(A)", "flying(A)"], "bird(A)"),**

**]**

**fc = ForwardChaining(rules)**

**fc.facts.add("mammal(duck)")**

**fc.facts.add("flying(duck)")**

**fc.run()**

**print(fc.facts)**

**if \_\_name\_\_ == "\_\_main\_\_":**

**main()**

**EXPLANATION:**

* The program defines a **Rule class** that represents a forward chaining rule. A rule has a list of premises and a conclusion.
* The **evaluate() method** checks if all of the premises of the rule are true in the given set of facts.
* The **fire() method** adds the conclusion of the rule to the set of facts.
* The ForwardChaining class represents a forward chaining inference engine. It has a list of rules and a set of facts.
* The **run() method** repeatedly evaluates all of the rules and fires any rules whose premises are true. The process stops when no more rules can be fired.
* The **main() function** creates a set of rules and a ForwardChaining object. It then adds some facts to the ForwardChaining object and runs the run() method. Finally, it prints the set of facts.

**OUTPUT:**

**{'bird(duck)', 'mammal(duck)', 'vertebrate(duck)', 'animal(duck)'}**

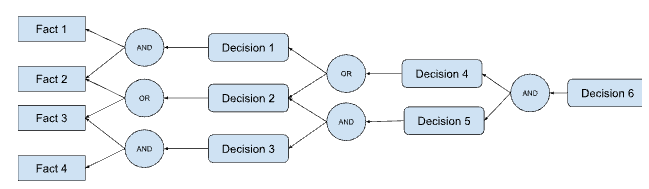
\*\*\* It shows that the forward chaining inference engine able to derive the fact that**duck is a bird from the facts that duck is a mammal and that duck can fly** \*\*\*

**EX.NO 4 b) IMPLEMENT BACKWARD CHAINING STRATEGIES**

**DEFINITION:**

**Backward chaining:** Backward chaining is another strategy used to shape an AI expert system that starts with the end goal and works backward through the AI’s rules to find facts that support the goal.

* Backward chaining is also known as a **backward deduction or backward reasoning method** when using an inference engine.
* In this, the inference engine knows the final decision or goal. The system starts from the goal and works backward to determine what facts must be asserted so that the goal can be achieved.
* For example, it starts directly with the conclusion (hypothesis) and validates it by backtracking through a sequence of facts.
* Backward chaining can be used in debugging, diagnostics and prescription applications.



**PROPERTIES:**

* Backward chaining uses an up-down strategy going from top to bottom.
* The modus ponens inference rule is used as the basis for the backward chaining process. This rule states that if both the conditional statement (p->q) and the antecedent (p) are true, then we can infer the subsequent (q).
* In backward chaining, the goal is broken into sub-goals to prove the facts are true.
* It is called a goal-driven approach, as a list of goals decides which rules are selected and used.
* The backward chaining algorithm is used in game theory, automated theorem-proving tools, inference engines, proof assistants and various AI applications.
* The backward-chaining method mostly used a depth-first search strategy for proof.

**PROGRAM:**

**class InferenceEngine:**

**def \_\_init\_\_(self, knowledge\_base):**

**self.knowledge\_base = knowledge\_base**

**def backward\_chaining(self, goal):**

# Check if the goal is already in the knowledge base

**if goal in self.knowledge\_base:**

**return True**

# Otherwise, search for a rule that can infer the goal

**for rule in self.knowledge\_base:**

**if rule.consequent == goal:**

# If the rule's antecedent is true, then the goal is true

**if self.backward\_chaining(rule.antecedent):**

**return True**

# If no rule can infer the goal, then the goal is false

**return False**

**class Rule:**

**def \_\_init\_\_(self, antecedent, consequent):**

**self.antecedent = antecedent**

**self.consequent = consequent**

# Create a knowledge base

**knowledge\_base = [**

**Rule("bird(x)", "can\_fly(x)"),**

**Rule("penguin(x)", "bird(x)"),**

**Rule("can\_swim(x)", "penguin(x)")**

**]**

# Create an inference engine

**inference\_engine = InferenceEngine(knowledge\_base)**

# Ask the inference engine if a penguin can fly

**can\_fly = inference\_engine.backward\_chaining("can\_fly(penguin)")**

**# Print the result**

**print(can\_fly)**

**OUTPUT:**

**False**

\*\*\* because penguins are birds, but they cannot fly \*\*\*