Experiment:-7

Objective:- Propositional Model Checking algorithm.

Theory:-Propositional logic is used for solving complex problems using simple statements. These statements can either be true or false but cannot be both at same time. These propositions form knowledge representation, reasoning and decision-making in AI systems. In this article we will see the basics of propositional logic and its applications in AI.
The core idea of the algorithm is to systematically evaluate the truth value of a propositional formula based on the truth values assigned to its atomic propositions within a specific model.
Example:
Consider the formula (P ∧ Q) → R and a model where P = True, Q = True, R = False. ● Evaluate P ∧ Q: Since P is True and Q is True, P ∧ Q is True.

- Evaluate (P ∧ Q) → R: The antecedent (P ∧ Q) is True, and the consequent R is False. According to the implication rule, True → False is False.
- Therefore, the formula (P ∧ Q) → R is False under this specific model.

Steps of the Algorithm:

- Identify Propositional Variables: Extract all distinct propositional variables (e.g., P, Q, R) present in the given propositional formula.
- Define the Model (Truth Assignment): A model is a mapping that assigns a truth value (True or False) to each of these propositional variables. This assignment represents a specific "state" or scenario.
- Evaluate Sub-formulas Recursively: The algorithm proceeds by recursively evaluating the truth value of sub-formulas, starting from the simplest components (literals) and working up to the entire formula.
    - Literals: If a sub-formula is a literal (a propositional variable or its negation), its truth value is directly obtained from the model. For example, if the model assigns True to P, then P is True and ¬P is False.
    - Logical Operators: For sub-formulas involving logical operators (∧, ∨, →, ↔, ¬), the truth value is determined based on the truth values of their operands according to the rules of propositional logic:
        - Negation (¬): If an operand is True, its negation is False, and vice versa.
        - Conjunction (∧): True only if both operands are True.
        - Disjunction (∨): True if at least one operand is True.
        - Implication (→): False only if the antecedent is True and the consequent is False.

● Biconditional (↔): True if both operands have the same truth value.

- Determine the Truth Value of the Entire Formula: After recursively evaluating all sub-formulas, the algorithm ultimately yields the truth value of the entire propositional formula under the given model.

Code:-

```
from itertools import product

def propositional_model_check(expression_str, valid_vars):
 variables = sorted(set(c for c in expression_str if c in valid_vars))
 truth_assignments = list(product([False, True], repeat=len(variables)))

 results = []
 for assignment in truth_assignments:
 env = dict(zip(variables, assignment))
 # In a real-world scenario, replace eval with a safer parser/evaluator
result = eval(expression_str, {}, env)
 results.append(result)

 if all(results):
 return "Tautology (always True)"
 elif any(results):
 return "Satisfiable but not a Tautology"
 else:
 return "Contradiction (always False)"

# Example
 expression = "(p or q) and (not p or q)"
valid_propositional_variables = {'p', 'q'}
 classification = propositional_model_check(expression, valid_propositional_variables)
print(f"The formula '{expression}' is: {classification}")
```

Output:-

```
The formula '(p or q) and (not p or q)' is: Satisfiable but not a Tautology
```

Experiment:-8

Objective:- Implement Forward Chaining algorithm.

Theory:- Forward chaining is a data-driven inference technique. It starts with the available data and applies rules to infer new data until a goal is reached. This method is commonly used in situations where the initial data set is extensive, and the goal is to derive conclusions from it.

How Forward Chaining Works

1. Start with Known Facts: The inference engine begins with the known facts in the knowledge base.
2. Apply Rules: It looks for rules whose conditions are satisfied by the known facts.
3. Infer New Facts: When a rule is applied, new facts are inferred and added to the knowledge base.
4. Repeat: This process is repeated until no more rules can be applied or a specified goal is achieved.

Example of Forward Chaining

Consider a medical diagnosis system where rules are used to diagnose diseases based on symptoms:

- Fact: The patient has a fever.
- Rule: If a patient has a fever and a rash, they might have measles.

Starting with the known fact (fever), the system checks for other symptoms (rash). If the patient also has a rash, the system infers the possibility of measles.

Advantages of Forward Chaining

1. Simplicity: Forward chaining is straightforward and easy to implement. 2. Automatic Data Processing: It processes data as it arrives, making it suitable for dynamic environments where new data continuously becomes available.
3. Comprehensive: It explores all possible inferences, ensuring that all relevant conclusions are reached.
4. Efficiency in Certain Scenarios: It can be efficient when all possible inferences need to be made from a set of data.

Disadvantages of Forward Chaining

1. Inefficiency in Goal-Oriented Tasks: It can be inefficient if only a specific goal needs to be achieved, as it may generate many irrelevant inferences.
2. Memory Intensive: It can consume significant memory, storing a large number of intermediate facts.
3. Complexity with Large Rule Sets: As the number of rules increases, the system may become slow due to the need to check many conditions.

Algorithm:-

1. Initialize the Knowledge Base:
    - Start with a set of known facts and a collection of if-then rules (production rules) in the knowledge base.
2. Identify Applicable Rules:
    - Scan the rules to find those whose conditions (antecedents) are satisfied by the

current set of facts.

3. Select a Rule to Fire:
   - If multiple rules are applicable, select one using a strategy (e.g., rule priority, order, or first-match). This is called conflict resolution.
4. Apply the Rule:
   - Execute the selected rule's action (consequent), which typically adds new facts to the knowledge base.
5. Update the Fact Base:
   - Incorporate any newly derived facts into the list of known facts.
6. Check for Goal or Termination Condition:
   - If the goal has been achieved or no more rules can fire, terminate the process. Otherwise, return to step 2.
7. Repeat Until Completion:
   - Continue the loop, applying applicable rules and expanding the fact base, until the desired conclusions are reached or no further inference is possible.

Code:-

```
class Rule:
 def __init__(self, antecedents, consequent):
 self.antecedents = set(antecedents) # Conditions that must be met
 self.consequent = consequent # Fact to be inferred if antecedents are met

 def __repr__(self):
 return f"IF {', '.join(self.antecedents)} THEN {self.consequent}"

def forward_chaining(facts, rules):
 """
 Implements the forward chaining algorithm to infer new facts.

 Args:
 facts (set): A set of initial known facts.
 rules (list): A list of Rule objects.

 Returns:
 set: The set of all inferred facts, including the initial facts.
 """
 inferred_facts = set(facts) # Start with the initial facts
 new_facts_inferred = True

 while new_facts_inferred:
 new_facts_inferred = False
```

```
 for rule in rules:
 # Check if all antecedents of the rule are present in inferred_facts  if
rule.antecedents.issubset(inferred_facts):
 # If the consequent is not already inferred, add it
 if rule.consequent not in inferred_facts:
 inferred_facts.add(rule.consequent)
 new_facts_inferred = True # Indicate that a new fact was inferred  return

inferred_facts

# Example Usage:
if __name__ == "__main__":
 # Define initial facts
 initial_facts = {"croaks", "eats_flies"}

 # Define rules
 rule1 = Rule(["croaks", "eats_flies"], "is_frog")
 rule2 = Rule(["is_frog"], "is_green")
 rule3 = Rule(["chirps", "sings"], "is_canary")
 rule4 = Rule(["is_canary"], "is_blue")

 knowledge_base_rules = [rule1, rule2, rule3, rule4]

 print("Initial Facts:", initial_facts)
 print("Rules in Knowledge Base:")
 for r in knowledge_base_rules:
 print(f"- {r}")

 # Run forward chaining
 final_facts = forward_chaining(initial_facts, knowledge_base_rules)

print("\nInferred Facts after Forward Chaining:", final_facts)



# Another example
 initial_facts_2 = {"is_seed", "is_mango"}
 rule5 = Rule(["is_seed"], "produces_plant")
 rule6 = Rule(["produces_plant", "is_mango"], "produces_fruit")
 knowledge_base_rules_2 = [rule5, rule6]

 print("\n--- Second Example ---")
 print("Initial Facts:", initial_facts_2)
 print("Rules in Knowledge Base:")
```

```
for r in knowledge_base_rules_2:
print(f"- {r}")
```

final_facts_2 = forward_chaining(initial_facts_2, knowledge_base_rules_2) print("\nInferred Facts after Forward Chaining:", final_facts_2)

Output:-

```
Initial Facts: {'croaks', 'eats_flies'}
Rules in Knowledge Base:
- IF croaks, eats_flies THEN is_frog
- IF is_frog THEN is_green
- IF sings, chirps THEN is_canary
- IF is_canary THEN is_blue

Inferred Facts after Forward Chaining: {'is_frog', 'is_green', 'croaks', 'eats_flies'}

--- Second Example ---
Initial Facts: {'is_mango', 'is_seed'}
Rules in Knowledge Base:
- IF is_seed THEN produces_plant
- IF is_mango, produces_plant THEN produces_fruit

Inferred Facts after Forward Chaining: {'is_mango', 'produces_fruit', 'is_seed',
    'produces_plant'}

=== Code Execution Successful ===
```

Experiment:-9

Objective:- Implement Backward Chaining algorithm.

Theory:- Backward chaining is a goal-driven inference technique. It starts with the goal and works backward to determine which facts must be true to achieve that goal. This method is ideal for situations where the goal is clearly defined, and the path to reach it needs to be established.

How Backward Chaining Works

1. Start with a Goal: The inference engine begins with the goal or hypothesis it wants to prove.
2. Identify Rules: It looks for rules that could conclude the goal.
3. Check Conditions: For each rule, it checks if the conditions are met, which may involve proving additional sub-goals.

4. Recursive Process: This process is recursive, working backward through the rule set until the initial facts are reached or the goal is deemed unattainable.

Example of Backward Chaining

In a troubleshooting system for network issues:

- Goal: Determine why the network is down.
- Rule: If the router is malfunctioning, the network will be down.

The system starts with the goal (network down) and works backward to check if the router is malfunctioning, verifying the necessary conditions to confirm the hypothesis.

Advantages of Backward Chaining

1. Goal-Oriented: It is efficient for goal-specific tasks as it only generates the facts needed to achieve the goal.
2. Resource Efficient: It typically requires less memory, as it focuses on specific goals rather than exploring all possible inferences.
3. Interactive: It is well-suited for interactive applications where the system needs to answer specific queries or solve particular problems.
4. Suitable for Diagnostic Systems: It is particularly effective in diagnostic systems where the goal is to determine the cause of a problem based on symptoms.

Disadvantages of Backward Chaining

1. Complex Implementation: It can be more complex to implement, requiring sophisticated strategies to manage the recursive nature of the inference process.
2. Requires Known Goals: It requires predefined goals, which may not always be feasible in dynamic environments where the goals are not known in advance.
3. Inefficiency with Multiple Goals: If multiple goals need to be achieved, backward chaining may need to be repeated for each goal, potentially leading to inefficiencies. 4. Difficulty with Large Rule Sets: As the number of rules increases, managing the backward chaining process can become increasingly complex.

Algorithm:-

1. Identify the Goal
   Start with the initial goal or query you want to prove. This could be a specific proposition or conclusion of interest.
2. Check if the Goal is Known

Determine if the goal already exists in the knowledge base as a fact. If it does, the goal is satisfied, and the algorithm returns success.

3. Find Applicable Rules

Search the knowledge base for rules whose conclusion (head) matches the current goal. These rules provide conditions (premises) that, if satisfied, can imply the goal. 4. Subdivide into Subgoals

For each applicable rule, list all its premises as subgoals that need to be satisfied. This creates a recursive structure where proving the current goal depends on proving its subgoals.

5. Recursively Apply Backward Chaining

Apply backward chaining to each subgoal in turn:

- If a subgoal is a known fact, mark it as satisfied.
- If not, attempt to prove it using rules in the knowledge base, generating further subgoals as needed.

6. Backtracking

If a rule fails (one or more subgoals cannot be satisfied), backtrack and try another applicable rule for that goal. Continue this process to explore all possibilities. 7. Determine Outcome

- If all subgoals for a rule are satisfied, the goal is satisfied.
- If no rules produce a satisfied goal after backtracking, the original goal cannot be proven given the current knowledge base.

8. Report Result

The algorithm concludes with either success (goal proven) or failure (goal cannot be

achieved). Optionally, the system can provide the inference path showing which rules and facts were used to reach the conclusion.

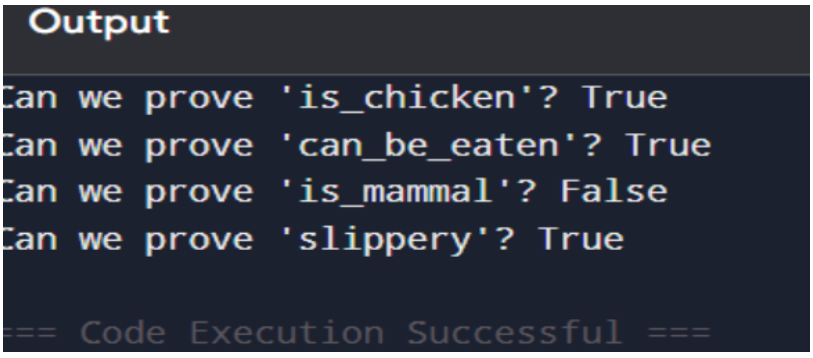
Code:-

```
class Rule:
 def __init__(self, antecedents, consequent):
 """
 Represents a rule in the knowledge base.
 :param antecedents: A list of conditions (strings) that must be true.
 :param consequent: The conclusion (string) that is true if all antecedents are true.  """
 self.antecedents = antecedents
 self.consequent = consequent

 def __repr__(self):
 return f"IF {self.antecedents} THEN {self.consequent}"
```

```
def backward_chaining(goal, facts, rules, inferred_facts=None):
 """
 Implements the backward chaining algorithm.
 :param goal: The goal (string) to be proven.
 :param facts: A set of known facts (strings).
 :param rules: A list of Rule objects in the knowledge base.
 :param inferred_facts: A set to store facts inferred during the process (for recursion).
:return: True if the goal can be proven, False otherwise.
 """
 if inferred_facts is None:
 inferred_facts = set()

 # 1. If the goal is already a known fact or has been inferred, it's proven.
if goal in facts or goal in inferred_facts:
 return True

 # 2. Find rules that conclude the goal.
 relevant_rules = [rule for rule in rules if rule.consequent == goal]

 for rule in relevant_rules:
 all_antecedents_proven = True
 for antecedent in rule.antecedents:
 # Recursively try to prove each antecedent

 if not backward_chaining(antecedent, facts, rules, inferred_facts):
all_antecedents_proven = False
 break

 # If all antecedents of a rule are proven, then the consequent (goal) is proven.  if
all_antecedents_proven:
 inferred_facts.add(goal) # Add the proven goal to inferred facts
 return True

 # If no rule can prove the goal, and it's not a fact, it cannot be proven.
 return False

# Example Usage:
if __name__ == "__main__":
 # Define facts
 known_facts = {"has_feathers", "lays_eggs", "eats_worms"}

 # Define rules
```

```
knowledge_base = [
Rule(["has_feathers", "lays_eggs"], "is_bird"),
Rule(["is_bird", "eats_worms"], "is_chicken"),
Rule(["is_chicken"], "is_poultry"),
Rule(["is_poultry"], "can_be_eaten")
]

# Test goals
goal1 = "is_chicken"
goal2 = "can_be_eaten"
goal3 = "is_mammal" # This goal cannot be proven with the given rules and facts

print(f"Can we prove '{goal1}'? {backward_chaining(goal1, known_facts, knowledge_base)}")
print(f"Can we prove '{goal2}'? {backward_chaining(goal2, known_facts, knowledge_base)}")
print(f"Can we prove '{goal3}'? {backward_chaining(goal3, known_facts, knowledge_base)}")

# Another example
facts_2 = {"rains"}
rules_2 = [

Rule(["rains"], "ground_is_wet"),
Rule(["ground_is_wet"], "slippery")
]
goal_4 = "slippery"
print(f"Can we prove '{goal_4}'? {backward_chaining(goal_4, facts_2, rules_2)}")
```

Output:-



Experiment:-10

Objective:- Implement Naive Bayes Models.

Theory:- Naive Bayes is a machine learning classification algorithm that predicts the category of a data point using probability. It assumes that all features are independent of each other. Naive Bayes performs well in many real-world applications such as spam filtering, document categorization and sentiment analysis.
The Naive Bayes algorithm is a supervised machine learning algorithm used for classification tasks. It is based on Bayes' Theorem and operates under a "naive" assumption of conditional independence among the features.

Key Concepts:

- Bayes' Theorem: This theorem allows calculating the probability of a hypothesis (e.g., an email being spam) given some evidence (e.g., the presence of certain words in the email). The formula is:

$$P(A|B) = P(B|A) * P(A) / P(B)$$

Where:

- P(A|B): Posterior probability of event A given event B.
- P(B|A): Likelihood of event B given event A.
- P(A): Prior probability of event A.
- P(B): Prior probability of event B.

How it Works:

To classify a new data point with a set of features, a Naive Bayes classifier calculates the posterior probability for each possible class. It does this by:

- Calculating Prior Probabilities: Determining the probability of each class in the training data.
- Calculating Likelihoods: Estimating the conditional probability of each feature given each class from the training data.
- Applying Bayes' Theorem: Multiplying the prior probability of each class by the likelihood of the observed features for that class.
- Classification: Assigning the new data point to the class with the highest calculated posterior probability.

Advantages:

- Simplicity and Efficiency: Easy to implement and computationally efficient, making it suitable for large datasets.
- Good Performance: Often delivers accurate results, especially in text classification and spam filtering.
- Interpretability: The contribution of each feature to a class prediction can be understood through conditional probabilities.

Limitations:

- Strong Independence Assumption: The "naive" assumption of feature independence is rarely true in real-world data, which can sometimes limit its accuracy.
- Zero-Frequency Problem: If a feature value is not observed in the training data for a particular class, its likelihood will be zero, causing the entire posterior probability for that class to become zero. Smoothing techniques (like Laplace smoothing) are used to address this.

Applications:

spam filtering, sentiment analysis, text classification, medical diagnosis, and fraud detection.

Code:-

```
import numpy as np
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.datasets import make_classification
```

```
# Generate a synthetic dataset
X, y = make_classification(n_samples=100, n_features=2, n_informative=2,
n_redundant=0, n_classes=2, random_state=42)

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Create a Gaussian Naive Bayes classifier
gnb = GaussianNB()

# Train the model
gnb.fit(X_train, y_train)


# Make predictions on the test set
y_pred = gnb.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Gaussian Naive Bayes Accuracy: {accuracy:.2f}")
```

Output:-