# Experiment No.-07

**Objective:** PROPOSITIONAL MODEL CHECKING ALGORITHMS .

## Theory:

In this method, the system is represented using propositional variables, each of which can take a truth value of either true or false. A propositional formula is a logical expression composed of these variables and logical connectives such as AND (∧), OR (∨), NOT (¬), IMPLIES (→), and IFF (↔). The model itself defines the truth assignment for each propositional variable, and model checking determines if the formula is satisfied under that assignment.

The main objective of propositional model checking is to evaluate if a given model satisfies a particular formula, denoted as M⊨ϕ M ⊨ φM⊨ϕ. The algorithm works recursively by evaluating the structure of the formula. If the formula is a single propositional variable, it checks the truth value assigned to it in the model. For compound formulas, it applies the logical rules: for negation, it inverts the truth value of the subformula; for conjunction and disjunction, it applies the corresponding truth tables; and for implication, it verifies that either the antecedent is false or the consequent is true. This systematic evaluation determines whether the overall formula evaluates to true or false in the given model.

For example, consider the formula (p∧q)→r(p ∧ q) → r(p∧q)→r and a model M={p=T,q=T,r=F}M = \{p = T, q = T, r = F\}M={p=T,q=T,r=F}. The expression p∧qp ∧ qp∧q evaluates to true, but since rrr is false, the implication (p∧q)→r(p ∧ q) → r(p∧q)→r evaluates to false. Therefore, the model does not satisfy the formula. Propositional model checking is widely used in verifying hardware circuits, software correctness, and communication protocols, ensuring that systems behave as expected. Although it is a powerful and automated verification method, it suffers from the state explosion problem, where the number of possible states grows exponentially with the system's complexity.

## Algorithm: Steps:

Step 1: Start.

Step 2: Input the propositional formula φφφ and the model MMM.

Step 3: If φφφ is a propositional variable, then:

- Return True if M(φ)=TM(φ) = TM(φ)=T.
- Else, return False.

Step 4: If φ=¬ψφ = ¬ψφ=¬ψ, then:

- Evaluate ModelCheck(M, ψ).
- Return NOT (ModelCheck(M, ψ)).

Step 5: If φ=ψ1∧ψ2φ = ψ₁ ∧ ψ₂φ=ψ1∧ψ2, then:
- Evaluate ModelCheck(M, ψ₁) and ModelCheck(M, ψ₂).
- Return (ModelCheck(M, ψ₁) AND ModelCheck(M, ψ₂)).

Step 6: If φ=ψ1∨ψ2φ = ψ₁ ∨ ψ₂φ=ψ1∨ψ2, then:
- Evaluate ModelCheck(M, ψ₁) and ModelCheck(M, ψ₂).
- Return (ModelCheck(M, ψ₁) OR ModelCheck(M, ψ₂)).

Step 7: If φ=ψ1→ψ2φ = ψ₁ → ψ₂φ=ψ1→ψ2, then:
- Evaluate ModelCheck(M, ψ₁) and ModelCheck(M, ψ₂).
- Return ((NOT ModelCheck(M, ψ₁)) OR ModelCheck(M, ψ₂)).

Step 8: Return the final truth value (True or False) indicating whether M⊨ϕM ⊨ ϕM⊨ϕ.

Step 9: Stop.

## Program:

```
def model_check(formula, model):
    """
    Function to evaluate a propositional formula in a given model.
    formula: string (e.g., "(p and q) -> r")
    model: dictionary assigning truth values to variables (e.g., {"p": True, "q": False, "r": True})
    """
    # Replace logical symbols with Python operators
    expr = formula.replace("¬", "not ").replace("∧", "and").replace("∨", "or").replace("→", "<=")
    expr = expr.replace("->", "<=").replace("AND", "and").replace("OR", "or").replace("NOT", "not")

    # Replace variable names with their truth values
    for var, val in model.items():
    expr = expr.replace(var, str(val))

    # Evaluate the final expression
    try:
    result = eval(expr)
    return result
    except:
```

```
    return "Invalid formula or model"


# --- Main Program ---

# Input propositional formula
formula = input("Enter propositional formula (use p, q, r and operators AND, OR, NOT, ->): ")

# Input model values
p_val = input("Enter value of p (True/False): ") == "True"
q_val = input("Enter value of q (True/False): ") == "True"
r_val = input("Enter value of r (True/False): ") == "True"

# Create model as dictionary
model = {"p": p_val, "q": q_val, "r": r_val}

# Perform model checking
result = model_check(formula, model)

# Display result
print("\nFormula:", formula)
print("Model:", model)
print("Result:", result)
```

**Output:**

```
Enter propositional formula (use p, q, r and operators AND, OR, NOT, ->): r
Enter value of p (True/False): True
Enter value of q (True/False): True
Enter value of r (True/False): False

Formula: r
Model: {'p': True, 'q': True, 'r': False}
Result: False

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

# Experiment No.-08

**8.1 Objective:**

To implement the Forward Chaining algorithm for an Expert System using a simple rule-based inference mechanism.

**8.2 Theory:**

Forward chaining is a **data-driven** inference technique used in expert systems.

- It starts from the **initial known facts**.
- It repeatedly checks which rules can fire (i.e., all conditions/antecedents of a rule are true).
- When a rule fires, its **consequent** is added as a new fact.
- This process continues until:
    - No new facts can be added, or
    - A particular **goal** fact is derived.

Forward chaining is useful in situations where **all input data is available** and we want to derive as many conclusions as possible from that data.

**8.3 Algorithm:**

1. Start with an initial set of known facts.
2. Read the list of production rules of the form:
   IF (conditions) THEN (conclusion).
3. Repeat the following steps:
    - For each rule:
        - Check if all its antecedents (conditions) are present in the fact base.
        - If yes, and its conclusion is **not already** in the fact base, then
            - Add the conclusion to the fact base.
            - Mark that some new fact was added.
4. Stop when:
    - No new fact is added in a complete pass over all rules, or
    - The required goal fact is found in the fact base.
5. Display the final set of facts.

**8.4 Program:**

```
# forward_chaining.py

# Simple Forward Chaining implementation (Expert System Example)

# Each rule is represented as: (list_of_antecedents, consequent)
rules = [
    (["fever", "cough"], "flu"),
    (["headache", "fever", "sore_throat"], "strep_throat"),
    (["sneezing", "runny_nose"], "cold"),
    (["flu", "body_ache"], "severe_flu"),
    (["cold", "fever"], "possible_flu")
]
def forward_chain(facts, rules, goal=None):
    facts = set(facts)   # fact base as a set
    print("Initial Facts:", facts)
    added = True
    while added:
        added = False
        for antecedents, consequent in rules:
            # if conclusion already known, skip
            if consequent in facts:
                continue
            # check if all antecedents are in facts
            if all(a in facts for a in antecedents):
                print(f"Rule fired: IF {antecedents} THEN
{consequent}")
                facts.add(consequent)
                added = True
```

```
                # if goal is given and achieved, we can stop
                if goal is not None and consequent == goal:
                    print(f"Goal '{goal}' achieved.")
                    print("Final Facts:", facts)
                    return facts

    print("No more rules can be fired.")
    print("Final Facts:", facts)
 if goal is not None:
        if goal in facts:
            print(f"Goal '{goal}' is in the fact base.")
        else:
            print(f"Goal '{goal}' could NOT be derived.")
    return facts
if __name__ == "__main__":
    # Example: initial facts
    initial_facts = ["fever", "cough", "body_ache"]
    goal = "severe_flu"
    forward_chain(initial_facts, rules, goal)
```

**8.5 Output:**

```
Goal 'severe_flu' achieved.
Final Facts: {'body_ache', 'fever', 'cough', 'severe_flu', 'flu'}


...Program finished with exit code 0
Press ENTER to exit console.
```

**8.6 Result:**

The Forward Chaining algorithm was successfully implemented and tested. The system was able to infer new facts (flu, severe_flu) from the given initial facts and rules.

# Experiment No.-09

**9.1 Objective:**

To implement the Backward Chaining algorithm for an Expert System to prove a goal using given rules and known facts.

**9.2 Theory:**

Backward chaining is a **goal-driven** inference technique.

- It starts from a **goal (hypothesis)** that we want to prove.
- It looks for rules that can conclude this goal.
- For each such rule, it tries to prove all its antecedents (sub-goals).
- This process continues recursively until either:
    - All sub-goals are satisfied using known facts, or
    - Some sub-goal cannot be satisfied (goal fails).

Backward chaining is commonly used in **diagnostic systems**, where we start with a hypothesis and check if the data supports it.

**9.3 Algorithm:**

1. Take the **goal** to be proved.
2. If the goal is present in the known fact list, return **TRUE**.
3. Otherwise, search for rules whose conclusion matches the goal.
4. For each such rule:
    - For every antecedent in that rule:
        - Recursively apply backward chaining to prove that antecedent.
    - If all antecedents are proved, then the goal is proved (TRUE).
5. If no rule can prove the goal, return **FALSE**.
6. Display whether the goal was successfully derived or not.

**9.4 Program:**

```
rules = {
    "flu": [["fever", "cough"], ["fever", "body_ache"]],
    "cold": [["sneezing", "runny_nose"]],
    "severe_flu": [["flu", "body_ache"]],
    "infection": [["fever", "high_temp"]]
}
def backward_chain(goal, known_facts, rules, indent=""):
```

```
    print(indent + f"Trying to prove: {goal}")
    # If goal already known
    if goal in known_facts:
        print(indent + f"Fact {goal} is known (given).")
        return True
    # If no rule concludes this goal
    if goal not in rules:
        print(indent + f"No rule to derive {goal}.")
        return False
    # Try each rule that can derive the goal
    for antecedent_list in rules[goal]:
print(indent + f"Checking rule: IF {antecedent_list} THEN
{goal}")
        all_proved = True
        for sub_goal in antecedent_list:
indent + "    ):
if not backward_chain(sub_goal, known_facts, rules,
                all_proved = False
                break
        if all_proved:
 print(indent + f"All conditions satisfied for {goal}. PROVED.")
            known_facts.add(goal)
            return Tru
    print(indent + f"Failed to prove: {goal}")
    return False
if __name__ == "__main__":
    # Known facts given by user
    # You can change this list for testing
    known_facts = {"fever", "cough", "body_ache"}
    goal = "severe_flu"
    print("Known Facts:", known_facts)
```

```
    print("Goal:", goal)
    result = backward_chain(goal, known_facts, rules)
    if result:
        print("\nFinal Result: Goal", goal, "is PROVED.")
    else:
        print("\nFinal Result: Goal", goal, "could NOT be proved.")
```

**9.5 Output:**

```
Known Facts: {'fever', 'body_ache', 'cough'}
Goal: severe_flu
Trying to prove: severe_flu
Checking rule: IF ['flu', 'body_ache'] THEN severe_flu
    Trying to prove: flu
    Checking rule: IF ['fever', 'cough'] THEN flu
        Trying to prove: fever
        Fact fever is known (given).
        Trying to prove: cough
        Fact cough is known (given).
    All conditions satisfied for flu. PROVED.
    Trying to prove: body_ache
    Fact body_ache is known (given).
All conditions satisfied for severe_flu. PROVED.

Final Result: Goal severe_flu is PROVED.


...Program finished with exit code 0
Press ENTER to exit console.
```

**9.6 Result:**

The Backward Chaining algorithm was successfully implemented and tested. The system was able to start from the goal and recursively prove it using the given rules and known facts.

# Experiment No.-10

## 10.1 Objective:

To implement a simple Naive Bayes classification model for an Expert System using a small training dataset.

## 10.2 Theory:

Naive Bayes is a probabilistic classifier based on Bayes' theorem with the assumption that all features are conditionally independent given the class.

Bayes theorem:

$$P(C \mid X) = \frac{P(X \mid C) \cdot P(C)}{P(X)}$$

Where:

- $C$ = class (e.g., disease)
- $X$ = observed features/symptoms
- $P(C \mid X)$ = posterior probability of class given features
- $P(C)$ = prior probability of class
- $P(X \mid C)$ = likelihood
- $P(X)$ = evidence (same for all classes, often ignored in comparison)

In Naive Bayes:

$$P(X \mid C) = \prod_i P(x_i \mid C)$$

because it assumes all features xix_ixi are independent given CCC.

The class with the highest posterior probability is chosen as the predicted class.

## 10.3 Algorithm:

1. Collect a small **training dataset** with:
   - Feature values (e.g., 'fever', 'cough', 'body_ache' = yes/no).
   - Class label (e.g., 'flu', 'cold').

2. Count occurrences to compute:
   - Prior probabilities P(C)P(C)P(C) for each class.
   - Conditional probabilities P(xi∣C)P(x_i \mid C)P(xi∣C) for each feature value and class.

3. For a new test instance:
   - For each class CCC:
     - Compute score = P(C)×∏P(xi∣C)P(C) \times \prod P(x_i \mid C)P(C)×∏P(xi∣C).
   - Choose the class with the **maximum score**.

4. Display the predicted class.

**10.4 Program:**

```
# naive_bayes.py
# Simple Naive Bayes model for disease classification (flu /
 cold)
# Training data: list of (features_dict, class_label)
# Features: fever, cough, body_ache (values: 'yes' or 'no')
training_data = [
"flu"),({"fever": "yes", "cough": "yes", "body_ache": "yes"},
    ({"fever": "yes", "cough": "yes", "body_ache": "no"}, "flu"),
"cold"),({"fever": "no", "cough": "yes", "body_ache": "no"},
"cold")({"fever": "no", "cough": "yes", "body_ache": "yes"},
"flu"),({"fever": "yes", "cough": "no", "body_ache": "yes"},
    ({"fever": "no", "cough": "no", "body_ache": "no"}, "cold")
]
def train_naive_bayes(data):
    class_counts = {}
    feature_counts = {}  # feature_counts[class][feature][value]
    total = len(data)
    for features, cls in data:
```

```
        # count classes
        class_counts[cls] = class_counts.get(cls, 0) + 1
        # count feature values per class
        if cls not in feature_counts:
            feature_counts[cls] = {}
        for f, v in features.items():
            if f not in feature_counts[cls]:
                feature_counts[cls][f] = {}
feature_counts[cls][f].get(v, 0) + 1feature_counts[cls][f][v]   =
    # calculate prior probabilities and conditional probabilities
(with simple Laplace smoothing)
    priors = {}
    cond_probs = {}

    for cls, c_count in class_counts.items():
        priors[cls] = c_count / total
        cond_probs[cls] = {}
        for f in feature_counts[cls]:
            cond_probs[cls][f] = {}
            # possible values assumed: 'yes' and 'no'
            for v in ["yes", "no"]:
                    # Laplace smoothing: +1 in numerator, +2 in
denominator
                count = feature_counts[cls][f].get(v, 0)
                 cond_probs[cls][f][v] = (count + 1) / (c_count +
2)

    return priors, cond_probs

def classify(features, priors, cond_probs):
    scores = {}
    for cls in priors:
        # start with prior
        score = priors[cls]
        for f, v in features.items():
```

```
            if f in cond_probs[cls]:
                score *= cond_probs[cls][f].get(v, 1)
        scores[cls] = score
    # choose class with maximum score
    best_class = max(scores, key=scores.get)
    return best_class, scores
if __name__ == "__main__":
    priors, cond_probs = train_naive_bayes(training_data)
    # Test instance: user can change these values
    test_features = {
        "fever": "yes",
        "cough": "yes",
        "body_ache": "yes"
    }
    predicted_class, scores = classify(test_features, priors,
cond_probs)
    print("Test Features:", test_features)
    print("Class Scores:", scores)
    print("Predicted Class:", predicted_class)
```

**10.5 Output:**

```
Class Scores: {'flu': 0.144, 'cold': 0.024}
Predicted Class: flu


...Program finished with exit code 0
Press ENTER to exit console.
```

## 10.6 Result:

The Naive Bayes model was successfully implemented. Using a small training dataset of symptoms and disease classes, the program correctly classified the test instance based on computed posterior probabilities.