Experiment:-1

Objective:- Implement Basic Search Strategies 8-Puzzle Problem.

Theory:- The 8-puzzle problem is a classic problem in artificial intelligence and search theory. It involves a 3x3 grid with 8 numbered tiles and one blank space. The objective is to move the tiles by sliding them into the blank space to reach a goal configuration from a given initial configuration.

Algorithm:-

Using BFS:- We can perform a Breadth-first search on the state space tree. This always finds a goal state nearest to the root. But no matter what the initial state is, the algorithm attempts the same sequence of moves like DFS.

- Breadth-first search on the state-space tree.
- Always finds the nearest goal state.
- Same sequence of moves irrespective of initial state.

Step by step approach:

- Start from the root node.
- Explore all neighboring nodes at the present depth.
- Move to the next depth level and repeat the process.
- If a goal state is reached, return the solution.

Program:-

```
from collections import deque
def bfs(start_state):
 target = [1, 2, 3, 4, 5, 6, 7, 8 , 0]
 dq = deque([start_state])
 visited = {tuple(start_state): None}
 while dq:
 state = dq.popleft()
 if state == target:
 path = []
 while state:
 path.append(state)
 state = visited[tuple(state)]
 return path[::-1]
 zero = state.index(0)
 row, col = divmod(zero, 3)
 for move in (-3, 3, -1, 1):
```

```
 new_row, new_col = divmod(zero + move, 3)
 if 0 <= new_row < 3 and 0 <= new_col < 3 and abs(row - new_row) + abs(col - new_col)
== 1:  neighbor = state[:]
 neighbor[zero], neighbor[zero + move] = neighbor[zero + move],
neighbor[zero] if tuple(neighbor) not in visited:
 visited[tuple(neighbor)] = state
 dq.append(neighbor)
def printSolution(path):
 for state in path:
 print("\n".join(' '.join(map(str, state[i:i+3])) for i in range(0, 9, 3)), end="\n-----\n")
```

 # Example Usage

```
startState = [1, 3, 0 , 6, 8, 4, 7, 5, 2]
solution = bfs(startState)
if solution:
 printSolution(solution)
 print(f"Solved in {len(solution) - 1} moves.")
else:
 print("No solution found.")
```

Output:-

```
 1 3 0
6 8 4
7 5 2
-----
 1 3 4
6 8 0
7 5 2
-----
 1 3 4
6 8 2
7 5 0
-----
 1 3 4
6 8 2
7 0 5
-----
.
.
-----
4 5 0
 7 8 6
-----
 1 2 3
4 5 6
```

7 8 0

Solved in 20 moves.

Using A*:-

A* is a computer algorithm that is widely used in pathfinding and graph traversal, the process of plotting an efficiently traversable path between multiple points, called nodes. Noted for its performance and accuracy, it enjoys widespread use. The A* algorithm is a heuristic search that combines aspects of both BFS and DFS. The A* algorithm uses heuristics to efficiently explore the state space and find an optimal solution faster than BFS or DFS.

Program:-

```
def a_star(start_state):
open_list = []
closed_list = set()
heapq.heappush(open_list, PuzzleState(start_state, None, None, 0,
heuristic(start_state)))  while open_list:
current_state = heapq.heappop(open_list)

if current_state.board == goal_state:
return current_state
closed_list.add(tuple(current_state.board))
blank_pos = current_state.board.index(0)
for move in moves:
if move == 'U' and blank_pos < 3: # Invalid move up
continue
if move == 'D' and blank_pos > 5: # Invalid move down
continue
if move == 'L' and blank_pos % 3 == 0: # Invalid move left
continue
if move == 'R' and blank_pos % 3 == 2: # Invalid move right
continue


new_board = move_tile(current_state.board, move, blank_pos)
if tuple(new_board) in closed_list:
continue
new_state = PuzzleState(new_board, current_state, move, current_state.depth
+ 1, current_state.depth + 1 + heuristic(new_board))
heapq.heappush(open_list, new_state)
return None
# Function to print the solution path
def print_solution(solution):
path = []
```

```
 current = solution
 while current:
 path.append(current)
 current = current.parent
 path.reverse()
 for step in path:
 print(f"Move: {step.move}")
 print_board(step.board)
# Initial state of the puzzle
initial_state = [1, 2, 3, 4, 0, 5, 6, 7, 8]
# Solve the puzzle using A* algorithm
 solution = a_star(initial_state)
# Print the solutionI
 f solution:
 print(colored("Solution found:", "green"))
 print_solution(solution)
else:
 print(colored("No solution exists.", "red"))
```

Output:-

Solution found:

Move: None

```
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 4 | | 5 |
+---+---+---+
| 6 | 7 | 8 |
+---+---+---+
```

Move: R

```
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 4 | 5 | |
+---+---+---+
| 6 | 7 | 8 |
+---+---+---+
.
.
```

Move: R

```
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 4 | 5 | 6 |
```

```
+---+---+---+
| 7 | | 8 |
+---+---+---+
```

Move: R

```
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 4 | 5 | 6 |
+---+---+---+
| 7 | 8 | |
+---+---+---+
```

Experiment:-2

Objective:- Implement Basic Search Strategies 8-queens Problem.

Theory:-

The **8-Queens problem** is a classic **constraint satisfaction problem (CSP)** in artificial intelligence and computer science. It involves placing **8 queens** on a standard **8×8 chessboard** such that **no two queens attack each other**. This means:

- No two queens share the same **row**
- No two queens share the same **diagonal**

Algorithm:-

Step by step approach:

1. Initialize an empty 8×8 chessboard with all positions set to 0.
2. Start with the first row (row 0) and try placing a queen in each column. 3. For each placement, check if the position is safe (not attacked by any previously placed queen). A position is unsafe if another queen is in the same column or on the same diagonal. 4. If a safe position is found, place the queen (set position to 1) and recursively try to place queens in subsequent rows. Otherwise, backtrack by removing the queen and trying the next column.
5. If all rows are successfully filled (8 queens placed), a valid solution is

found. **Program:-**

```
def printSolution(board):
 """Print the chessboard configuration."""
 for row in board:
 print(" ".join("Q" if col else "." for col in row))
 print("\n")

def isSafe(board, row, col, n):
 """Check if placing a queen at board[row][col] is safe."""
 # Check column


 for i in range(row):
 if board[i][col]:
 return False

 # Check upper-left diagonal
 i, j = row, col
 while i >= 0 and j >= 0:
```

```
if board[i][j]:
return False
i -= 1

    j -= 1
# Check upper-right diagonal
i, j = row, col
while i >= 0 and j < n:
if board[i][j]:
return False
i -= 1
j += 1

return True

def solveNQueens(board, row, n):
"""Use backtracking to solve the N-Queens problem."""
if row == n:
printSolution(board)
return True

result = False
for col in range(n):
if isSafe(board, row, col, n):
# Place the queen
board[row][col] = 1
# Recur to place the rest of the queens
result = solveNQueens(board, row + 1, n) or result
# Backtrack
board[row][col] = 0

return result

def nQueens(n):
"""Driver function to solve the N-Queens problem."""
board = [[0] * n for _ in range(n)]
if not solveNQueens(board, 0, n):
print("No solution exists.")
else:
print("Solutions printed above.")

# Solve the 8-Queens problem
nQueens(8)
```

Output:-

Q . . . . . .
. . . . Q . . .
. . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . Q .
. Q . . . . . .
. . . Q . . . .

Q . . . . . .
. . . . . Q . .
. . . . . Q
. . Q . . . . .
. . . . . Q .
. . . Q . . . .
. Q . . . . . .
. . . . Q . . .

. . . . . Q
. . . Q . . . .
Q . . . . . .
. . Q . . . . .
. . . . . Q . .
. Q . . . . . .
. . . . . Q .
. . . . Q . . .

Experiment:-3

Objective:- Implementation of Basic Search strategies-CRYPT ARITHEMATIC.

Theory:- Cryptarithmetic (also known as verbal arithmetic or alphametics) is a type of mathematical puzzle where the digits in an arithmetic equation are replaced by letters. The objective is to find the correct digit each letter represents so that the arithmetic equation is valid.

Solving these puzzles involves search strategies—systematic ways of exploring possible letter-digit assignments.

General Rules:

1. Each alphabet takes only one number from 0 to 9 uniquely.
2. Two single digit numbers sum can be maximum 19 with carryover. So carry over in problems of two number addition is always 1.
3. Try to solve the left most digit in the given problem.
4. If a × b = kb, then the following are the possibilities
(3 × 5 = 15; 7 × 5 = 35; 9 × 5 = 45) or (2 × 6 = 12; 4 × 6 = 24; 8 × 6 = 48)

Algorithm:-

The idea is to firstly create a list of all the characters that need assigning to pass to Solve. Now use backtracking to assign all possible digits to the characters.

- If all characters are assigned, return true if puzzle is solved, false otherwise ● Otherwise, consider the first unassigned character
- for (every possible choice among the digits not in use)
- If all digits have been tried and nothing worked, return false to trigger backtracking

Program:-

```
import itertools

def solve_cryptarithmetic(equation):
 """
 Solves a cryptarithmetic puzzle.
 e.g., "SEND + MORE = MONEY"
 """
 # Split the equation into left-hand side (LHS) and right-hand side (RHS)
lhs_str, rhs_str = equation.lower().replace(' ', '').split('=')
 lhs_words = lhs_str.split('+')

 # Collect all unique letters in the puzzle
 letters = set()
 for char in word:
 letters.add(char)
```

```
for char in rhs_str:
letters.add(char)
letters = list(letters)

# Identify the first letters of each word (cannot be zero)
first_letters = {word[0] for word in lhs_words}.union({rhs_str[0]})

# Generate all possible permutations of digits for the letters
digits = range(10)
for perm in itertools.permutations(digits, len(letters)):
mapping = dict(zip(letters, perm))

# Check if any first letter is mapped to zero
if any(mapping[char] == 0 for char in first_letters):
continue

# Evaluate the LHS and RHS with the current mapping
try:
lhs_value = sum(word_to_int(word, mapping) for word in
lhs_words)  rhs_value = word_to_int(rhs_str, mapping)

# If the equation holds true, print the solution
if lhs_value == rhs_value:
print(f"Solution found for '{equation}':")
for letter, digit in mapping.items():
print(f" {letter} = {digit}")
print(f" {lhs_value} = {rhs_value}")
return True # Found a solution, exit
except KeyError:
# This can happen if a letter in the equation isn't in 'letters',
# but our setup ensures all letters are included.
pass

print(f"No solution found for '{equation}'.")
return False

def word_to_int(word, mapping):
"""Converts a word to its integer value based on the letter-digit mapping."""

value = 0
for char in word:
value = value * 10 + mapping[char]
return value
```

```
# Example usage:
solve_cryptarithmetic("SEND + MORE = MONEY")
```

Output:-

```
SEND: 9567, MORE: 1085, MONEY: 10652

Solution: {'S': 9, 'E': 5, 'N': 6, 'D': 7, 'M': 1, 'O': 0, 'R': 8, 'Y': 2}

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

Experiment:-4

Objective:-IMPLEMENT A* ALGORITHM

Theory:- The A* algorithm is a highly effective and well-known search technique utilized for finding the most efficient path between two points in a graph. It is applied in scenarios such as pathfinding in video games, network routing and various artificial intelligence (AI) applications. It was developed in 1968 by Peter Hart, Nils Nilsson and Bertram Raphael as an improvement on Dijkstra's algorithm.

Key Components of A* Algorithm

A* uses two important parameters to find the cost of a path:

1.g(n): Actual cost of reaching node n from the start node. This is the accumulated cost of the path from the start node to node n.

2.h(n): The heuristic finds the cost to reach the goal from node n . This is a weighted guess about how much further it will take to reach the goal.

The function, f(n)=g(n)+h(n) is the total estimated cost of the cheapest solution through node n. This function combines the path cost so far and the heuristic cost to estimate the total cost guiding the search more efficiently.

To understand A* algorithm, you need to be familiar with these fundamental concepts:

- **Nodes**: Points in your graph (like intersections on a map)
- **Edges**: Connections between nodes (like roads connecting intersections) ● **Path Cost**: The actual cost of moving from one node to another
- **Heuristic**: An estimated cost from any node to the goal
- **Search Space**: The collection of all possible paths to explore

Program:-

```
import heapq
def heuristic(a, b):

 return abs(a[0] - b[0]) + abs(a[1] - b[1])

def a_star_search(grid, start, goal):

 rows, cols = len(grid), len(grid[0])


 open_set = []
heapq.heappush(open_set, (0, start))
 came_from = {}
```

```
 g_score = {node: float('inf') for row in range(rows) for node in [(row, c) for c in
range(cols)]}  g_score[start] = 0

 f_score = {node: float('inf') for row in range(rows) for node in [(row, c) for c in
range(cols)]}  f_score[start] = heuristic(start, goal)

 while open_set:
 current_f_cost, current_node = heapq.heappop(open_set)

 if current_node == goal:

 path = []
 while current_node in came_from:
 path.append(current_node)
 current_node = came_from[current_node]
 path.append(start)
 return path[::-1]

 for dr, dc in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
 neighbor = (current_node[0] + dr, current_node[1] + dc)

 if 0 <= neighbor[0] < rows and \
 0 <= neighbor[1] < cols and \
 grid[neighbor[0]][neighbor[1]] == 0:
 tentative_g_score = g_score[current_node] + 1

 if tentative_g_score < g_score[neighbor]:
 came_from[neighbor] = current_node
 g_score[neighbor] = tentative_g_score
 f_score[neighbor] = g_score[neighbor] + heuristic(neighbor, goal)
heapq.heappush(open_set, (f_score[neighbor], neighbor))

 return None

if __name__ == "__main__":
 grid = [
[0, 0, 0, 0, 0],
 [0, 1, 0, 1, 0],
 [0, 1, 0, 0, 0],
 [0, 0, 0, 1, 0],
 [0, 0, 0, 0, 0]
 ]
 start_node = (0, 0)
 goal_node = (4, 4)
 path = a_star_search(grid, start_node, goal_node)
```

```
if path:
print("Path found:")
 for r, c in path:
print(f"({r}, {c})")
 else:
print("No path found.")
```

Output:-

```
Path found:
(0, 0)
(0, 1)
(0, 2)
(0, 3)
(0, 4)
(1, 4)
(2, 4)
(3, 4)
(4, 4)
```

Experiment No.-05

Objective: Implement MINI MAX Algorithm for game playing(ALPHA-BETA PRUNING)**.**

Theory:- The MiniMax algorithm with Alpha-Beta Pruning is an optimization of the basic MiniMax algorithm, used in game theory and artificial intelligence to determine the optimal move for a player in a two-player, zero-sum game.
Conceptual Overview:

- **Minimax Principle:** The algorithm explores the game tree, assuming both players play optimally. The maximizing player (Max) aims to maximize their score, while the minimizing player (Min) aims to minimize Max's score (which is equivalent to maximizing their own score in a zero-sum game).
- **Alpha-Beta Pruning:** This optimization prunes branches of the game tree that are guaranteed not to affect the final decision. It maintains two values:
- **Alpha (α):** The best (highest) value that the maximizing player can guarantee so far.
- **Beta (β):** The best (lowest) value that the minimizing player can guarantee so far.
- If at any point beta <= alpha, it means that a better move has already been found for the current player's opponent in a different branch, making the current branch irrelevant. The search can be terminated for that branch.

Program:-

```
import math

def minimax_alpha_beta(node_index, depth, is_maximizing_player, scores, alpha, beta):

   # Base case: if at a leaf node (terminal state)     if
depth == 0:  # Assuming depth 0 means a terminal node
return scores[node_index]

   if is_maximizing_player:
max_eval = -math.inf
      # Simulate moves (assuming 2 children per node for simplicity)
for i in range(2):
         child_index = 2 * node_index + i + 1  # Calculate child index          eval
= minimax_alpha_beta(child_index, depth - 1, False, scores, alpha, beta)
max_eval = max(max_eval, eval)          alpha = max(alpha, eval)          if beta
<= alpha:
```

```
        break  # Alpha-Beta
Pruning        return max_eval
else:
      min_eval =
math.inf       #
Simulate moves
for i in range(2):
          child_index = 2 * node_index + i + 1           eval =
minimax_alpha_beta(child_index, depth - 1, True, scores, alpha, beta)
min_eval = min(min_eval, eval)           beta = min(beta, eval)           if beta <=
alpha:               break  # Alpha-Beta Pruning        return min_eval

# Example scores for leaf nodes (indices 3, 4, 5, 6 in a conceptual tree)
leaf_scores = [3, 5, 2, 9] # These are the scores at the deepest level of the tree.

full_scores_list = [0] * 7 # Placeholder for non-leaf
nodes full_scores_list[3] = 3 full_scores_list[4] = 5
full_scores_list[5] = 2 full_scores_list[6] = 9

optimal_value = minimax_alpha_beta(0, 2, True, full_scores_list, -math.inf, math.inf)
print(f"The optimal value is: {optimal_value}")
```

Output:-

```
The optimal value is: 3
```

Experiment No.-06

Objective: Solve Constraint Satisfaction Problems.

Theory: A Constraint Satisfaction Problem (CSP) is a mathematical problem defined as a set of objects whose state must satisfy a number of constraints or limitations.

It is widely used in Artificial Intelligence, operations research, and optimization.

A CSP is represented by:

• X = {X1, X2, ..., Xn} → a set of variables

• D = {D1, D2, ..., Dn} → domains of possible values for each variable

• C = {C1, C2, ..., Ck} → a set of constraints specifying allowable combinations of values

A solution to a CSP is an assignment of values to all variables that satisfies all constraints.

➢ Types of CSPs:

• Discrete CSPs: Variables take values from a finite domain (e.g., Sudoku, N-Queens).

• Continuous CSPs: Variables have continuous domains (e.g., scheduling problems).

• Dynamic CSPs: Constraints or variables may change over time.

➢ Common Techniques for Solving CSPs:

• Backtracking Search: Systematically explores possible variable assignments and backtracks when constraints are violated.

• Forward Checking: Prevents future conflicts by checking consistency during assignment.

• Arc Consistency (AC-3 Algorithm): Removes inconsistent values from domains.

• Heuristics:

o Minimum Remaining Values (MRV) — choose the variable with the fewest remaining legal values.

o Least Constraining Value (LCV) — choose the value that rules out the fewest options for neighboring variables.

➢ Applications:

• Map coloring

• Sudoku solving

• Scheduling and timetabling

• Resource allocation

• N-Queens problem

Algorithm: Steps:

Step 1: Initialization

• Define the set of variables and their possible domains.

• Define the constraints between variables.

Step 2: Selection

• Choose an unassigned variable.

Step 3: Assignment
• Assign a value from its domain.
Step 4: Constraint Checking
• Check if the assignment is consistent with the constraints.
Step 5: Backtrack
• If a constraint is violated, undo the last assignment and try a different value.
Step 6: Termination
• If all variables are assigned valid values, output the solution.
• If no valid assignment exists, report failure.
Time Complexity:
Worst case — O(dn) where d = size of domain, n = number of variables.
Space Complexity:
O(n) for recursion stack and variable assignments.

Program:

```
def is_safe(assignment, var, value, constraints):
for neighbor in constraints[var]:
if neighbor in assignment and assignment[neighbor] == value:
return False
return True

def backtrack(variables, domains, constraints, assignment):
if len(assignment) == len(variables):
return assignment # All variables assigned

var = [v for v in variables if v not in assignment][0]

for value in domains[var]:
if is_safe(assignment, var, value, constraints):
assignment[var] = value
result = backtrack(variables, domains, constraints, assignment)
if result:
return result
del assignment[var] # backtrack
return None

def main():
variables = ['WA', 'NT', 'SA', 'Q', 'NSW', 'V', 'T']

domains = {
'WA': ['Red', 'Green', 'Blue'],
'NT': ['Red', 'Green', 'Blue'],
'SA': ['Red', 'Green', 'Blue'],
'Q': ['Red', 'Green', 'Blue'],
```

```
'NSW': ['Red', 'Green', 'Blue'],
'V': ['Red', 'Green', 'Blue'],
'T': ['Red', 'Green', 'Blue']
}

constraints = {
'WA': ['NT', 'SA'],
'NT': ['WA', 'SA', 'Q'],
'SA': ['WA', 'NT', 'Q', 'NSW', 'V'],
'Q': ['NT', 'SA', 'NSW'],
'NSW': ['Q', 'SA', 'V'],
'V': ['SA', 'NSW'],
'T': []
}

assignment = {}
solution = backtrack(variables, domains, constraints, assignment)

if solution:
print("Solution Found:")
for state, color in solution.items():
print(f"{state} → {color}")
else:
print("No solution exists.")

if __name__ == "__main__":
main()
```

Output:

```
Solution Found:
WA → Red
NT → Green
SA → Blue
Q → Red
NSW → Green
V → Red
T → Red
```

Experiment No.-07

Objective: PROPOSITIONAL MODEL CHECKING ALGORITHMS .

Theory:

Propositional model checking is an automatic technique used to verify whether a given propositional logic formula holds true within a particular model or state. 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 $M \models \phi$ 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. Despite this limitation, propositional model checking remains a foundational concept in formal verification and serves as the basis for more advanced techniques like temporal and symbolic model checking.

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, ψ_1) and ModelCheck(M, ψ_2).
- Return (ModelCheck(M, ψ_1) AND ModelCheck(M, ψ_2)).

Step 6: If φ=ψ1∨ψ2φ = ψ_1 ∨ ψ2φ=ψ1∨ψ2, then:

- Evaluate ModelCheck(M, ψ_1) and ModelCheck(M, ψ_2).
- Return (ModelCheck(M, ψ_1) OR ModelCheck(M, ψ_2)).

Step 7: If φ=ψ1→ψ2φ = ψ_1 → ψ2φ=ψ1→ψ2, then:

- Evaluate ModelCheck(M, ψ_1) and ModelCheck(M, ψ_2).
- Return ((NOT ModelCheck(M, ψ_1)) OR ModelCheck(M, ψ_2)).

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 ===
```