LAB-9: FOL(forward chaining),Min-max(tic-tac-toe), Alpha-Beta Pruning(8-Queens)

Observation book:

classmate
Date 03/12/24
Page ______

First Order Logic [Forward Chaining]:

"As per the law, it is a crime for an American to sell weapons to hostile nations. Country A, an enemy of America, has some missiles, ~~was sold~~ and all the missiles were sold to it by Robert, who is an American citizen"

Prove that "Robert is criminal"

Predicates:

* American (x) : x is an American citizen
* Hostile (x, y) : x is a hostile nation to y
* Sold (x, m, y) : x sold missile m to y
* Criminal (x) : x is a criminal

~~Logical~~ Logical Axioms:

* It is a crime for an American to sell weapons to hostile nations.

- $\forall x\, \forall m\, \forall y\, (\text{American}(x) \wedge \text{Hostile}(y, \text{America}) \wedge \text{Sold}(x, m, y) \rightarrow \text{Criminal}(x))$

* Robert is an American.

- American (Robert)

* Country A is hostile to America.

- Hostile (A, America)

* Robert sold missiles to country A:

- $\exists m$ Sold (Robert, m, A)

Forward Chaining:

* Using the above facts, we can use forward chaining to combine them and arrive at:

American (Robert) $\wedge$ Hostile (A, America) $\wedge$ Sold (Robert, m, A) $\longrightarrow$ Criminal (Robert).

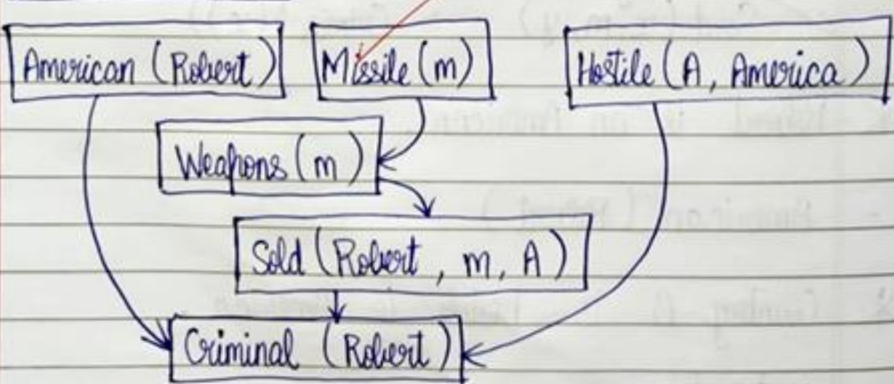
Conclusion:

By forward chaining, we can conclude that "Robert is a criminal"

Note:

* If question says forward chaining with proof, show each derivation:
- Missile $(x) \Rightarrow$ Weapon (x)
- Draw the derivation tree level by level.

Derivation Tree:



Date 3/12/2024
Page

Min-Max Algorithm (Tic-Tac-Toe) :

* Minimizing the possible loss for a worst-case scenario by maximizing the player's minimum gain.

Algorithm :

```
                  → board  → player
Function check(b, p)
  for i in range(3) do
      if all(board[i][j] == p for j in range(3)) do
          return True
      end if
      if all(board[j][i] == p for j in range(3)) do
          return True
      end if
  end for
  if all(board[i][i] == p for i in range(3)) do
     return True
  end if
  if all(board[i][2-i] == p for i in range(3)) do
     return True
  end if
  return False

Function FullBoard(board)
  for r in board do
      if " " in r do
          return False
      end if
  end for
  return True
```

<P.T.O>

```
Function Evaluate (b)
  if check (b, X) do
    return 1
  end if
  if check (b, O) do
    return -1
  end if
  return 0

Function minmax (b, d, flag)
  score ← evaluate (b)
  if score == 1 or score == -1 or FullBoard(b) do
    return score
  end if
  if flag do
    best ← - float ('inf')
    for i in range (3) do
      for j in range (3) do
        if b[i][j] == " " do
          b[i][j] ← "X"
          best ← max (best, minmax (b, d+1, False))
          b[i][j] ← " "
        end if
      end for
    end for
    return best
  else do
    best ← float ('inf')
    for i in range (3) do
      for j in range (3) do
        if b[i][j] == " " do
          b[i][j] ← "O"
```

```
                best ← min (best, minmax (b, d+1, True))
                b[i][j] ← " "
            end if
        end for
    end for
    return best
end if

Function Find_best_move (b)
    best_val ← -float ('inf')
    best_move ← (-1, -1)
    for i in range (3) do
        for j in range (3) do
            if b[i][j] = EMPTY do
                b[i][j] ← X
                move_val ← minmax (b, 0, False)
                b[i][j] ← Empty
                if move_val > best_val do
                    best_move ← (i, j)
                    best_val ← move_val
                end if
            end if
        end for
    end for
    return best_move

Function print_board (b)
    for r in bo do
        print (" | ".join(r))
        print ("-" * 5)
```

<P·T·O>

```
if _name_ == _main_ do
    b ← [[ EMPTY, EMPTY, EMPTY],
         [ EMPTY, EMPTY, EMPTY],
         [ EMPTY, EMPTY, EMPTY]]
    print ("Initial Board : ")
    print_board (b)
    while True do
        best_move ← Find_best_move (b)
        print (Player X plays <best_move>)
        b[best_move[0]][best_move[1]] ← X
        print_board (b)
        if Evaluate (b) = 1 do
            print (Player X wins)
            break
        elif is_board_full (b)
            print (It's a draw)
            break
        import random
        empty_positions ← [(i, j) for i in range(3) for
                            j in range(3) if b[i][j] =
                            EMPTY]
        if empty_positions do
            o_move ← random. choice (empty_positions)
            print (Player O plays <o_move> )
            b[o_move[0]][o_move[1]] ← O
            print_board (b)
            if evaluate (b) = -1 do
                print (Player O wins)
                break
            elif is_board_full (b) do
                print ("Its a draw")
                break
```

Alpha - Beta Pruning (8- Queens):

```
Function is_safe (b, r, c)
  for i from 0 to r-1 do
     if b[i] = c or abs(b[i]-c) = abs(i-r) do
        return False
     end if
  end for
  Return True

Function alpha_beta (b, r, α, β)
   if r = 8 do
      if is_solution (b)
          print b
          Return True
      end if
      Return False
   end if
   for c from 0 to 7 do
      if is_safe (b, r, c) do
          b[r] ← c
          if alpha_beta (b, r+1, α, β) do
              Return True
          end if
          b[r] ← -1
      end if
      if r % 2 = 0 do
          α ← max (α, value)
      else do
          β ← min (β, value)
      end if
      if α >= β do
```

break
end if
end for
Return False

Function is_solution (b)
Return (no. of distinct values in board == 8)

Function main ()
b ← [-1, -1, -1, -1, -1, -1, -1, -1]
$\alpha \leftarrow -\infty$
$\beta \leftarrow -\infty$
alpha_beta (b, 0, α, β)

30/12

Code:

FOL (Forward Chaining)

```
class Fact:
    def __init__(self, predicate, *args):
        self.predicate = predicate
        self.args = tuple(args)

    def __eq__(self, other):
        return self.predicate == other.predicate and self.args == other.args

    def __hash__(self):
        return hash((self.predicate, self.args))

    def __str__(self):
        return f"{self.predicate}({', '.join(self.args)})"

class Rule:
    def __init__(self, conditions, conclusion):
        self.conditions = conditions  # A list of Facts
        self.conclusion = conclusion  # A single Fact

    def is_satisfied(self, known_facts):
        return all(condition in known_facts for condition in self.conditions)

    def __str__(self):
        conditions_str = " ^ ".join(str(c) for c in self.conditions)
        return f"{conditions_str} -> {self.conclusion}"

class ForwardChaining:
```

```
    def __init__(self):
        self.facts = set()  # Set of known facts
        self.rules = []  # List of rules

    def add_fact(self, fact):
        self.facts.add(fact)

    def add_rule(self, rule):
        self.rules.append(rule)

    def infer(self):
        new_facts = True
        while new_facts:
            new_facts = False
            for rule in self.rules:
                if rule.is_satisfied(self.facts) and rule.conclusion not in self.facts:
                    # Printing the logical statement applied when the rule is applied
                    print(f"Applying rule: {rule.conditions} -> {rule.conclusion}")
                    self.facts.add(rule.conclusion)
                    new_facts = True

    def display_facts(self):
        print("\nFinal Set of Statements proving that Robert is a criminal:")
        for fact in self.facts:
            print(f"{fact.predicate.capitalize()} of {', '.join(fact.args)} is true.")

if __name__ == "__main__":
    fc = ForwardChaining()
```

```
    # Hardcoding facts as per the problem statement
    fc.add_fact(Fact("crime", "american", "hostile_nation"))  # It is a crime for an American to sell weapons to a hostile nation
    fc.add_fact(Fact("american", "robert"))  # Robert is an American
    fc.add_fact(Fact("sold_missiles", "robert", "country_a"))  # Robert sold missiles to Country A
    fc.add_fact(Fact("enemy", "country_a", "america"))  # Country A is an enemy of America

    # Rule: If an American sells weapons to a hostile nation, they are a criminal
    conditions = [
        Fact("american", "robert"),
        Fact("sold_missiles", "robert", "country_a"),
        Fact("enemy", "country_a", "america")
    ]
    conclusion = Fact("criminal", "robert")
    fc.add_rule(Rule(conditions, conclusion))

    # Perform inference (forward chaining)
    print("Performing inference...\n")
    fc.infer()

    # Display the results: final set of facts proving Robert is a criminal
    fc.display_facts()
print("Nikhilesh C – 1BM22CS181")
```

Output:

```
Performing inference...

Applying rule: [<__main__.Fact object at 0x000001E54819B640>, <__main__.Fact obj
ect at 0x000001E54819B6D0>, <__main__.Fact object at 0x000001E54819BA60>] -> cri
minal(robert)

Final Set of Statements proving that Robert is a criminal:
Enemy of country_a, america is true.
Crime of american, hostile_nation is true.
Sold_missiles of robert, country_a is true.
Criminal of robert is true.
American of robert is true.
Nikhilesh C - 1BM22CS181
```

MINIMAX (TIC-TAC-TOE):

Code:

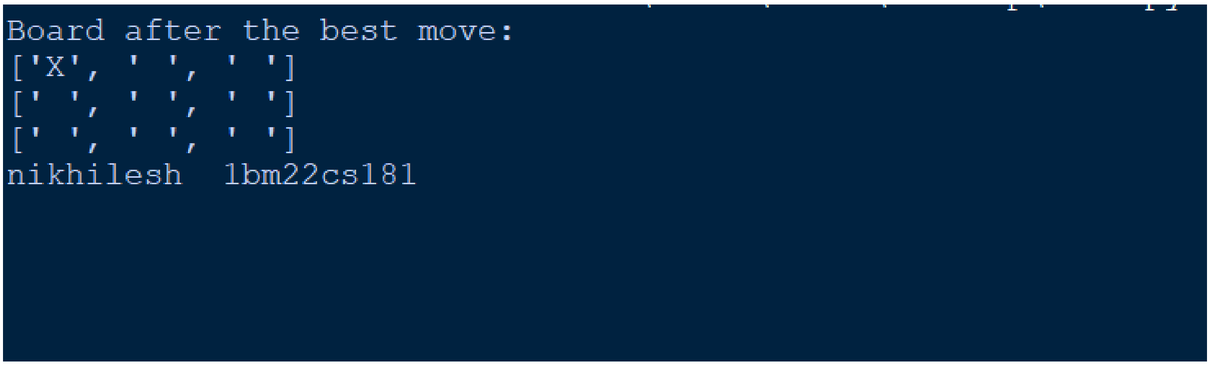
```
import math


def minimax(board, depth, is_maximizing_player):
    if game_over(board):
        return evaluate(board)


    if is_maximizing_player:
        best = -math.inf
        for move in available_moves(board):
            make_move(board, move, 'X')
            best = max(best, minimax(board, depth + 1, False))
            undo_move(board, move)
        return best
    else:
        best = math.inf
        for move in available_moves(board):
            make_move(board, move, 'O')
            best = min(best, minimax(board, depth + 1, True))
            undo_move(board, move)
        return best
```

```
def evaluate(board):
    if player_wins(board, 'X'):
        return 1
    if player_wins(board, 'O'):
        return -1
    return 0

def game_over(board):
    return player_wins(board, 'X') or player_wins(board, 'O') or no_more_moves(board)

def available_moves(board):
    moves = []
    for row in range(3):
        for col in range(3):
            if board[row][col] == " ":
                moves.append((row, col))
    return moves

def make_move(board, move, player):
    row, col = move
    board[row][col] = player

def undo_move(board, move):
    row, col = move
    board[row][col] = " "

def player_wins(board, player):
    # Check rows and columns
```

```
    for i in range(3):
        if all(board[i][j] == player for j in range(3)) or all(board[j][i] == player for j in range(3)):
            return True
    # Check diagonals
    if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for i in range(3)):
        return True
    return False

def no_more_moves(board):
    return all(board[row][col] != " " for row in range(3) for col in range(3))

def main():
    board = [[" " for _ in range(3)] for _ in range(3)]
    current_player = 'X'
    best_move = None

    if current_player == 'X':
        best_score = -math.inf
        for move in available_moves(board):
            make_move(board, move, 'X')
            score = minimax(board, 0, False)
            undo_move(board, move)
            if score > best_score:
                best_score = score
                best_move = move
        make_move(board, best_move, 'X')

    print("Board after the best move:")
    for row in board:
```

```
        print(row)


if __name__ == "__main__":
    main()
print("nikhilesh  1bm22cs181")
```

Output :



```
Board after the best move:
['X', ' ', ' ']
[' ', ' ', ' ']
[' ', ' ', ' ']
nikhilesh  1bm22cs181
```

Alpha-beta(8 Queens)

Code:

```
# Function to check if placing a queen at (row, col) is safe
def is_safe(board, row, col):
    for i in range(row):
        if board[i] == col or abs(board[i] - col) == abs(i - row):  # Check for column and diagonal
conflicts
            return False
    return True


# Backtracking function for N-Queens
def solve_n_queens(board, row):
    if row == 8:  # All queens have been placed
        print_board(board)  # Print the board if solution is found
        return True
```

```
    for col in range(8):  # Try placing a queen in each column of the current row
        if is_safe(board, row, col):  # Check if placing a queen at (row, col) is safe
            board[row] = col  # Place the queen in the current column

            # Recursively attempt to place the next queen in the next row
            if solve_n_queens(board, row + 1):
                return True  # Solution found, propagate up

            board[row] = -1  # Backtrack: Remove the queen from the current position

    return False  # No solution found in the current row and column configurations

# Function to print the board in a readable format
def print_board(board):
    for row in range(8):
        line = ['Q' if board[row] == col else '.' for col in range(8)]
        print(" ".join(line))
    print()

# Main function to start solving the N-Queens problem
def main():
    board = [-1] * 8  # Initialize the board (no queens placed)
    if not solve_n_queens(board, 0):  # Start solving from the first row
        print("No solution found.")

# Call the main function
main()

print("Nikhilesh 1BM22CS181")
```

Output:

```
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .

Nikhilesh 1BM22CS181
>>
```