Siddharth.KP
241501203

Program Debug Debug Result Python ...

Fill your code here

```python
import copy

N = 8

def printSolution(board):
    for row in board:
        for i in range(N):
            print("Q" if row[i] else ".", end=" ")
        print()
    print()

def isSafe(board, row, col):
    for i in range(row):
        if board[i][col]:
            return False
    for i, j in zip(range(row - 1, -1, -1), range(col - 1, -1, -1)):
        if board[i][j]:
            return False
    for i, j in zip(range(row - 1, -1, -1), range(col + 1, N)):
        if board[i][j]:
            return False
    return True

def solve(board, row, solutions):
    if row == N:
        solutions.append(copy.deepcopy(board))
        printSolution(board)
        return
    for col in range(N):
```

```python
def solve(board, row, solutions):
    if row == N:
        solutions.append(copy.deepcopy(board))
        printSolution(board)
        return
    for col in range(N):
        if isSafe(board, row, col):
            board[row][col] = 1
            solve(board, row + 1, solutions)
            board[row][col] = 0

def eightQueens():
    board = [[0 for _ in range(N)] for _ in range(N)]
    solutions = []
    solve(board, 0, solutions)
    print(f"Total solutions found: {len(solutions)}")

eightQueens()
```

Siddharth.KP
241501203

Compiler Message

```
Compilation successful
```

Custom Testcase

Output

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

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

Program Debug Debug Result Python ...

Fill your code here

```python
warehouse_graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

def dfs(graph, start, goal, visited=None, path=None):
    if visited is None:
        visited = set()
    if path is None:
        path = []

    visited.add(start)
    path.append(start)

    if start == goal:
        return path

    for neighbor in graph[start]:
        if neighbor not in visited:
            result = dfs(graph, neighbor, goal, visited, path[:])
            if result:
                return result
    return None

start_node = 'A'
goal_node = 'F'

path_found = dfs(warehouse_graph, start_node, goal_node)
print(f"DFS Path from {start_node} to {goal_node}: {path_found}")
```

Siddharth.KP
241501203

```python
start_node = 'A'
goal_node = 'F'

path_found = dfs(warehouse_graph, start_node, goal_node)
print(f"DFS Path from {start_node} to {goal_node}: {path_found}")

```

Clear Compile & Run

Enter your custom input

Compiler Message

Compilation successful

Custom Testcase

Output

```
DFS Path from A to F: ['A', 'B', 'E', 'F']
```

Program Debug Debug Result Python ...

Fill your code here

```python
PLAYER_X = 1
PLAYER_O = -1
EMPTY = 0

def evaluate(board):
    for row in range(3):
        if board[row][0] == board[row][1] == board[row][2] != EMPTY:
            return board[row][0]
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] != EMPTY:
            return board[0][col]
    if board[0][0] == board[1][1] == board[2][2] != EMPTY:
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != EMPTY:
        return board[0][2]
    return 0

def isMovesLeft(board):
    for row in range(3):
        for col in range(3):
            if board[row][col] == EMPTY:
                return True
    return False

def minimax(board, isMax):
    score = evaluate(board)
    if score == PLAYER_X:
        return score
    if score == PLAYER_O:
        return score
    if not isMovesLeft(board):
        return 0
```

Siddharth.KP
241501203

Fill your code here

```
            return 0

        if isMax:
            best = -float('inf')
            for row in range(3):
                for col in range(3):
                    if board[row][col] == EMPTY:
                        board[row][col] = PLAYER_X
                        best = max(best, minimax(board, not isMax))
                        board[row][col] = EMPTY
            return best
        else:
            best = float('inf')
            for row in range(3):
                for col in range(3):
                    if board[row][col] == EMPTY:
                        board[row][col] = PLAYER_O
                        best = min(best, minimax(board, not isMax))
                        board[row][col] = EMPTY
            return best

def findBestMove(board):
    bestVal = -float('inf')
    bestMove = (-1, -1)
    for row in range(3):
        for col in range(3):
            if board[row][col] == EMPTY:
                board[row][col] = PLAYER_X
                moveVal = minimax(board, False)
                board[row][col] = EMPTY
                if moveVal > bestVal:
                    bestMove = (row, col)
```

Fill your code here

```
def findBestMove(board):
    bestVal = -float('inf')
    bestMove = (-1, -1)
    for row in range(3):
        for col in range(3):
            if board[row][col] == EMPTY:
                board[row][col] = PLAYER_X
                moveVal = minimax(board, False)
                board[row][col] = EMPTY
                if moveVal > bestVal:
                    bestMove = (row, col)
```


```
                    bestVal = moveVal
    return bestMove

def printBoard(board):
    for row in board:
        print(" ".join(["X" if x == PLAYER_X else "O" if x == PLAYER_

board = [
    [PLAYER_X, PLAYER_O, PLAYER_X],
    [PLAYER_O, PLAYER_X, EMPTY],
    [EMPTY, PLAYER_O, PLAYER_X]
]

print("Current Board:")
printBoard(board)
move = findBestMove(board)
print(f"Best Move: {move}")
board[move[0]][move[1]] = PLAYER_X
print("\nBoard after best move:")
printBoard(board)
```

Siddharth.KP
241501203

Compiler Message

```
Compilation successful
```

Custom Testcase

Output

```
Current Board:
X O X
O X .
. O X
Best Move: (1, 2)

Board after best move:
X O X
O X X
. O X
```

Fill your code here

```
import heapq

class Node:
    def __init__(self, position, parent=None, g=0, h=0):
        self.position = position
        self.parent = parent
        self.g = g
        self.h = h
        self.f = g + h

    def __lt__(self, other):
        return self.f < other.f

def heuristic(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def a_star(grid, start, goal):
    rows, cols = len(grid), len(grid[0])
    open_list = []
    heapq.heappush(open_list, Node(start, None, 0, heuristic(start, go
    closed_set = set()

    while open_list:
        current_node = heapq.heappop(open_list)

        if current_node.position == goal:
            path = []
            while current_node:
                path.append(current_node.position)
                current_node = current_node.parent
            return path[::-1]

```

Siddharth.KP
241501203

```python
        if current_node.position == goal:
            path = []
            while current_node:
                path.append(current_node.position)
                current_node = current_node.parent
            return path[::-1]

        closed_set.add(current_node.position)

        for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            new_pos = (current_node.position[0] + dr, current_node.p
            if (0 <= new_pos[0] < rows and 0 <= new_pos[1] < cols ar
                grid[new_pos[0]][new_pos[1]] == 0 and new_pos not i
                new_node = Node(new_pos, current_node, current_node.
                                heuristic(new_pos, goal))
                heapq.heappush(open_list, new_node)

    return None

warehouse_grid = [
    [0, 0, 0, 0, 1],
    [1, 1, 0, 1, 0],
    [0, 0, 0, 0, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 0, 0]
]

start_position = (0, 0)
goal_position = (4, 4)
path = a_star(warehouse_grid, start_position, goal_position)
print("Optimal Path:", path)
```

Compiler Message

```
Compilation successful
```

Custom Testcase

Output

```
Optimal Path: [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2, 3), (2, 4), (3, 4), (4, 4)]
```

Siddharth.KP
241501203

Program Debug Debug Result Python ...

Fill your code here

```
class Person:
    def __init__(self, name):
        self.name = name
        self.likes = set()

mary = Person("mary")
john = Person("john")

mary.likes.update(["food", "wine"])
john.likes.update(["wine", "mary"])

def infer_likes(john, mary):
    for item in mary.likes:
        john.likes.add(item)
    if "wine" in [x for x in mary.likes] or "wine" in [x for x in john.
        john.likes.add("mary")
        john.likes.add("john")
    if "wine" in mary.likes:
        john.likes.add("wine")

infer_likes(john, mary)

print("Who does John like?")
for item in john.likes:
    print(item)
```

Compiler Message

```
Compilation successful
```

Custom Testcase

Output

```
Who does John like?
food
mary
wine
john
```

Siddharth.KP
241501203

Program Debug Debug Result

```python
import re

# Function to check if two predicates can be unified
def unify(x, y, theta={}):
    if theta is None:
        return None
    elif x == y:
        return theta
    elif isinstance(x, str) and x.islower():  # x is a variable
        return unify_var(x, y, theta)
    elif isinstance(y, str) and y.islower():  # y is a variable
        return unify_var(y, x, theta)
    elif isinstance(x, list) and isinstance(y, list) and len(x) == len(y):
        return unify(x[1:], y[1:], unify(x[0], y[0], theta))
    else:
        return None

# Function to unify a variable with a term
def unify_var(var, x, theta):
    if var in theta:
        return unify(theta[var], x, theta)
    elif x in theta:
        return unify(var, theta[x], theta)
    else:
        theta[var] = x
        return theta

# Function to apply resolution rule
def resolution(kb, query):
    for clause in kb:
        theta = unify(clause[0], query, {})
        if theta is not None:
            new_kb = clause[1:]
            if not new_kb:  # If empty, means query is resolved
                return True
            else:
                return resolution(kb, new_kb[0])
    return False

# Knowledge base (Implications)
knowledge_base = [
    [["Human", "John"], ["Mortal", "John"]],  # Human(John) → Mortal(John)
]

# Fact: Human(John)
fact = ["Human", "John"]

# Query: Mortal(John)?
query = ["Mortal", "John"]

# Apply resolution
if resolution(knowledge_base, query):
    print("Query is resolved: John is Mortal")
else:
    print("Query could not be resolved")
```

Siddharth.KP
241501203

Compiler Message

```
Compilation successful
```

Custom Testcase

Output

```
Query could not be resolved
```

Program Debug Debug Result

```python
# Knowledge Base (Rules in IF-THEN format)
knowledge_base = {
    "flu": [["cough", "fever"]],
    "fever": [["sore_throat"]],
}

# Known facts
facts = {"sore_throat", "cough"}

# Backward chaining function
def backward_chaining(goal):
    """
    Determines if the goal can be inferred from known facts and the knowledge base.

    Args:
        goal (str): The condition to be checked (e.g., 'flu').

    Returns:
        bool: True if the goal can be inferred, False otherwise.
    """
    if goal in facts:
        return True
    if goal in knowledge_base:
        for conditions in knowledge_base[goal]:
            if all(backward_chaining(cond) for cond in conditions):
                return True
    return False

# Query: Does the patient have flu?
query = "flu"
if backward_chaining(query):
    print(f"The patient is diagnosed with {query}.")
else:
    print(f"The patient does NOT have {query}.")
```

Siddharth.KP
241501203

Compiler Message

```
Compilation successful
```

Custom Testcase

Output

```
The patient is diagnosed with flu.
```

Program Debug Debug Result

```python
# Knowledge Base: Rules in IF-THEN format
knowledge_base = [
    (["cough", "fever"], "flu"),
    (["sore_throat", "runny_nose"], "cold"),
    (["sore_throat"], "fever")  # Sore throat can lead to fever
]

# Given initial facts
facts = {"cough", "sore_throat"}

# Forward Chaining Function
def forward_chaining():
    inferred = True  # Keep looping as long as new facts are added
    while inferred:
        inferred = False  # Stop if no new fact is added in an iteration
        for conditions, conclusion in knowledge_base:
            if all(condition in facts for condition in conditions) and conclusion not in facts:
                facts.add(conclusion)  # Add the inferred fact
                inferred = True  # Mark that we inferred a new fact

# Run forward chaining
forward_chaining()

# Check if flu or cold is inferred
if "flu" in facts:
    print("The patient is diagnosed with flu.")
elif "cold" in facts:
    print("The patient is diagnosed with cold.")
else:
    print("No conclusive diagnosis could be made.")
```

Siddharth.KP
241501203

Compiler Message

```
Compilation successful
```

Custom Testcase

Output

```
The patient is diagnosed with flu.
```

Siddharth.KP
241501203