main.py Share Run Output

```python
N = 8
def print_solution(board):
    for row in board:
        print(" ".join("Q" if cell else "." for cell in row))
    print()

def is_safe(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):
    if row == N:
        print_solution(board)
        return True
    for col in range(N):
```

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


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

```python
    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):
    if row == N:
        print_solution(board)
        return True
    for col in range(N):
        if is_safe(board, row, col):
            board[row][col] = 1
            if solve(board, row + 1):
                return True
            board[row][col] = 0
    return False
board = [[0 for _ in range(N)] for _ in range(N)]

if not solve(board, 0):
    print("No solution found.")
```

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


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

```python
#Depth first algorithm
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()

    visited.add(start)
    print(start, end=' ')

    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
print("Depth-First Search starting from A:")
dfs(graph, 'A')
```

```
Depth-First Search starting from A:
A B D E F C
=== Code Execution Successful ===
```

```python
#Program for A*
import heapq

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_set = []
    heapq.heappush(open_set, (0, start))

    came_from = {}
    g_score = {start: 0}
    f_score = {start: heuristic(start, goal)}

    while open_set:
        current_f, current = heapq.heappop(open_set)

        if current == goal:
            path = []
            while current in came_from:
                path.append(current)
```

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

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

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

    for dx, dy in [(-1,0),(1,0),(0,-1),(0,1)]:
        neighbor = (current[0] + dx, current[1] + dy)

        if 0 <= neighbor[0] < rows and 0 <= neighbor[1] < cols:
            if grid[neighbor[0]][neighbor[1]] == 1:
                continue
            tentative_g = g_score[current] + 1
            if neighbor not in g_score or tentative_g <
                g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score[neighbor] = tentative_g + heuristic
                    (neighbor, goal)
                heapq.heappush(open_set, (f_score[neighbor],
                    neighbor))
```

```python
                        (neighbor, goal)
                    heapq.heappush(open_set, (f_score[neighbor],
                        neighbor))

    return None
grid = [
    [0, 0, 0, 0, 0],
    [1, 1, 0, 1, 0],
    [0, 0, 0, 1, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 0, 0]
]
start = (0, 0)
goal = (4, 4)
path = a_star(grid, start, goal)
if path:
    print("Path found:")
    for step in path:
        print(step)
else:
    print("No path found.")
```

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

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

```python
class AOStar:     #Program for AO*
    def __init__(self, graph, heuristic):
        self.graph = graph
        self.heuristic = heuristic
        self.status = {}
        self.solution = {}

    def get_min_cost_child_nodes(self, node):
        if node not in self.graph:
            return 0, []

        min_cost = float('inf')
        best_group = []

        for group in self.graph[node]:
            cost = 0
            for child in group:
                cost += self.heuristic[child]
            if cost < min_cost:
                min_cost = cost
                best_group = group

```

```
Expanding Node: A
Expanding Node: D
Expanding Node: G
Expanding Node: D

Solution Path:
A -> D -> G


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

```python
            best_group = group

    return min_cost, best_group

def ao_star(self, node, backtracking=False):
    print(f"Expanding Node: {node}")

    if node not in self.graph or not self.graph[node]:
        self.status[node] = 'Solved'
        return

    cost, best_group = self.get_min_cost_child_nodes(node)
    self.heuristic[node] = cost
    self.solution[node] = best_group

    all_solved = True
    for child in best_group:
        if self.status.get(child) != 'Solved':
            all_solved = False
            self.ao_star(child, backtracking=True)

    if all_solved:
```

```
Expanding Node: A
Expanding Node: D
Expanding Node: G
Expanding Node: D

Solution Path:
A -> D -> G

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

```python
        if all_solved:
            self.status[node] = 'Solved'

        if backtracking:
            self.ao_star(node, backtracking=False)

    def print_solution(self, node):
        if node not in self.solution or not self.solution[node]:
            print(node, end='')
            return
        print(node, end=' -> ')
        children = self.solution[node]
        for i, child in enumerate(children):
            self.print_solution(child)
            if i != len(children) - 1:
                print(" & ", end='')
graph = {
    'A': [['B', 'C'], ['D']],
    'B': [['E'], ['F']],
    'C': [['G']],
    'D': [['G']],
    'E': [],
```

```
Expanding Node: A
Expanding Node: D
Expanding Node: G
Expanding Node: D

Solution Path:
A -> D -> G

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

```python
graph = {
    'A': [['B', 'C'], ['D']],
    'B': [['E'], ['F']],
    'C': [['G']],
    'D': [['G']],
    'E': [],
    'F': [],
    'G': []
}
heuristic = {
    'A': 10,
    'B': 4,
    'C': 4,
    'D': 2,
    'E': 3,
    'F': 2,
    'G': 0
}
aostar = AOStar(graph, heuristic)
aostar.ao_star('A')
print("\nSolution Path:")
aostar.print_solution('A')
print()
```

```
Expanding Node: A
Expanding Node: D
Expanding Node: G
Expanding Node: D

Solution Path:
A -> D -> G

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

```python
import math
PLAYER_X = 'X'
PLAYER_O = 'O'
EMPTY = ' '
def print_board(board):
    for row in board:
        print('|'.join(row))
        print('-' * 5)
def is_winner(board, player):
    for row in board:
        if all(s == player for s in row):
            return True
    for col in range(3):
        if all(board[row][col] == player for row in range(3)):
            return True
    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 is_full(board):
    return all(cell != EMPTY for row in board for cell in row)

```

```
 | | 
-----
 | | 
-----
 | | 
-----
Enter your move (row and col 0-2): 1 1
 | | 
-----
 |X| 
-----
 | | 
-----
AI's move:
O| | 
-----
 |X| 
-----
 | | 
-----
Enter your move (row and col 0-2): 0 0
Cell is already occupied! Try again.
```

```python
        return max_eval
    else:
        min_eval = math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == EMPTY:
                    board[i][j] = PLAYER_X
                    eval = minimax(board, depth + 1, True, alpha, beta
                        )
                    board[i][j] = EMPTY
                    min_eval = min(min_eval, eval)
                    beta = min(beta, eval)
                    if beta <= alpha:
                        break
        return min_eval
def best_move(board):
    best_val = -math.inf
    move = (-1, -1)
    for i in range(3):
        for j in range(3):
            if board[i][j] == EMPTY:
                board[i][j] = PLAYER_O
```

```
Enter your move (row and col 0-2): 0 0
Cell is already occupied! Try again.
O| |
-----
 |X|
-----
 | |
-----
Enter your move (row and col 0-2): 0 2
O| |X
-----
 |X|
-----
 | |
-----
AI's move:
O| |X
-----
 |X|
-----
O| |
-----
```

```python
                if board[i][j] == EMPTY:
                    board[i][j] = PLAYER_O
                    move_val = minimax(board, 0, False, -math.inf, math
                        .inf)
                    board[i][j] = EMPTY
                    if move_val > best_val:
                        best_val = move_val
                        move = (i, j)
    return move
def play_game():
    board = [[EMPTY for _ in range(3)] for _ in range(3)]
    current_player = PLAYER_X

    while True:
        print_board(board)
        if current_player == PLAYER_X:
            row, col = map(int, input("Enter your move (row and col 0
                -2): ").split())
            if board[row][col] != EMPTY:
                print("Cell is already occupied! Try again.")
                continue
        else:
```

```
-----
 |X|
-----
O| |
-----
Enter your move (row and col 0-2): 2 2
O| |X
-----
 |X|
-----
O| |X
-----
AI's move:
O| |X
-----
O|X|
-----
O| |X
-----
AI wins!
```

```python
        else:
            print("AI's move:")
            row, col = best_move(board)

        board[row][col] = current_player

        if is_winner(board, current_player):
            print_board(board)
            if current_player == PLAYER_X:
                print("You win!")
            else:
                print("AI wins!")
            break

        if is_full(board):
            print_board(board)
            print("It's a draw!")
            break

        current_player = PLAYER_X if current_player == PLAYER_O else PLAYER_O

```

```
AI's move:
O| |X
-----
 |X|
-----
O| |
-----
Enter your move (row and col 0-2): 2 2
O| |X
-----
 |X|
-----
O| |X
-----
AI's move:
O| |X
-----
O|X|
-----
O| |X
-----
AI wins!
```

```python
        board[row][col] = current_player

        if is_winner(board, current_player):
            print_board(board)
            if current_player == PLAYER_X:
                print("You win!")
            else:
                print("AI wins!")
            break

        if is_full(board):
            print_board(board)
            print("It's a draw!")
            break

        current_player = PLAYER_X if current_player == PLAYER_O else PLAYER_O


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

```
AI's move:
O| |X
-----
 |X|
-----
O| |
-----
Enter your move (row and col 0-2): 2 2
O| |X
-----
 |X|
-----
O| |X
-----
AI's move:
O| |X
-----
O|X|
-----
O| |X
-----
AI wins!
```

Python Online Compiler

₹75 000
www.hyundai.co.in

main.py

Share

Run

```python
# 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:
```

Output

```
Query could not be resolved

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

Light rain
At night

Search

ENG
IN

main.py

Share Run

```python

# 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
```

Output

```
Query could not be resolved

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

Search

main.py

Share Run

```python
# 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)
```

Output

```
Query could not be resolved

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

```
    [["Human", "John"], ["Mortal", "John"]],  # Human(John) →
        Mortal(John)
]

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

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

# Add the fact to the knowledge base
knowledge_base.append([fact])  # Facts as unit clauses

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

Output

```
Query could not be resolved

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

Classroom Management Tools | Sam's POAI AE & AF | Online Python Compiler (Interp | Introducing ChatGPT | OpenAI

programiz.com/python-programming/online-compiler/

Programiz

Python Online Compiler

Premium Coding Courses by Programiz

main.py

Share Run

```python
knowledge_base = {
    "flu": [["cough", "fever"]],
    "fever": [["sore_throat"]],
}

facts = {"sore_throat", "cough"}

def backward_chaining(goal):
    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 = "flu"
if backward_chaining(query):
    print(f"The patient is diagnosed with {query}.")
else:
    print(f"The patient does NOT have {query}.")
```

Output

```
The patient is diagnosed with flu.

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

gramiz
on Online Compiler

Premium Coding
Courses by Programiz

ain.py

Share Run

Output

```python
import re

def unify(x, y, theta={}):
    if theta is None:
        return None
    elif x == y:
        return theta
    elif isinstance(x, str) and x.islower():
        return unify_var(x, y, theta)
    elif isinstance(y, str) and y.islower():
        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

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
```

```
Query could not be resolved

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

ssroom Management Tools × | Sam's PDAI AE & AF × | Online Python Compiler (Inter × | Introducing ChatGPT | OpenAI ×

programiz.com/python-programming/online-compiler/

gramiz
on Online Compiler

Premium Coding
Courses by Programiz

main.py | Share | Run | Output

```python
        return theta

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:
                return True
            else:
                return resolution(kb, new_kb[0])
    return False

knowledge_base = [
    [["Human", "John"], ["Mortal", "John"]],
]

fact = ["Human", "John"]

query = ["Mortal", "John"]

if resolution(knowledge_base + [[fact]], query):
    print("Query is resolved: John is Mortal")
else:
    print("Query could not be resolved")
```

```
Query could not be resolved

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

Programiz

Python Online Compiler

Premium Coding Courses by Programiz

main.py

Share Run

```python
knowledge_base = [
    (["cough", "fever"], "flu"),
    (["sore_throat", "runny_nose"], "cold"),
    (["sore_throat"], "fever")
]

facts = {"cough", "sore_throat"}

def forward_chaining():
    inferred = True
    while inferred:
        inferred = False
        for conditions, conclusion in knowledge_base:
            if all(condition in facts for condition in conditions)
                and conclusion not in facts:
                facts.add(conclusion)
                inferred = True

forward_chaining()

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.")
```

Output

```
The patient is diagnosed with flu.

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

Programiz

Python Online Compiler

Premium Coding Courses by Programiz

Programiz PRO

main.py

Share Run

```python
class BlocksWorld:
    def __init__(self):
        self.state = {
            "A": "B",
            "B": "table",
            "C": "table"
        }
        self.goal = {
            "A": "B",
            "B": "C",
            "C": "table"
        }

    def is_goal_state(self):
        return self.state == self.goal

    def move(self, block, destination):
        if block in self.state and self.state[block] != destination:
            print(f"Moving {block} from {self.state[block]} to
                {destination}")
            self.state[block] = destination

    def plan_moves(self):
        print("\nInitial State:", self.state)
        while not self.is_goal_state():
            for block, target in self.goal.items():
```

Output

```
Initial State: {'A': 'B', 'B': 'table', 'C': 'table'}
Moving B from table to C

Final Goal State Reached: {'A': 'B', 'B': 'C', 'C': 'table'}

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

Programiz
Python Online Compiler

Premium Coding Courses by Programiz

Programiz PRO

main.py

Share Run

```python
        self.goal = {
            "A": "B",
            "B": "C",
            "C": "table"
        }

    def is_goal_state(self):
        return self.state == self.goal

    def move(self, block, destination):
        if block in self.state and self.state[block] != destination:
            print(f"Moving {block} from {self.state[block]} to
                {destination}")
            self.state[block] = destination

    def plan_moves(self):
        print("\nInitial State:", self.state)
        while not self.is_goal_state():
            for block, target in self.goal.items():
                if self.state[block] != target:
                    self.move(block, target)
        print("\nFinal Goal State Reached:", self.state)

bw = BlocksWorld()
bw.plan_moves()
```

Output

```
Initial State: {'A': 'B', 'B': 'table', 'C': 'table'}
Moving B from table to C

Final Goal State Reached: {'A': 'B', 'B': 'C', 'C': 'table'}

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

Programiz
Python Online Compiler

Premium Coding Courses by Programiz

main.py

Share Run

```python
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl

experience = ctrl.Antecedent(np.arange(0, 21, 1), 'experience')
success_rate = ctrl.Antecedent(np.arange(0, 101, 1), 'success_rate')
performance = ctrl.Consequent(np.arange(0, 101, 1), 'performance')

experience['low'] = fuzz.trimf(experience.universe, [0, 0, 10])
experience['medium'] = fuzz.trimf(experience.universe, [5, 10, 15])
experience['high'] = fuzz.trimf(experience.universe, [10, 20, 20])

success_rate['low'] = fuzz.trimf(success_rate.universe, [0, 0, 50])
success_rate['medium'] = fuzz.trimf(success_rate.universe, [25, 50, 75])
success_rate['high'] = fuzz.trimf(success_rate.universe, [50, 100, 100])

performance['poor'] = fuzz.trimf(performance.universe, [0, 0, 50])
performance['average'] = fuzz.trimf(performance.universe, [25, 50, 75])
performance['excellent'] = fuzz.trimf(performance.universe, [50, 100, 100])

rule1 = ctrl.Rule(experience['low'] & success_rate['low'], performance['poor'])
```

Output

```
predicted performance score : 67.85
```

Programiz
Python Online Compiler

Premium Coding
Courses by Programiz

main.py   Share   Run

```python
    100])

performance['poor'] = fuzz.trimf(performance.universe, [0, 0, 50])
performance['average'] = fuzz.trimf(performance.universe, [25, 50,
    75])
performance['excellent'] = fuzz.trimf(performance.universe, [50, 100
    , 100])

rule1 = ctrl.Rule(experience['low'] & success_rate['low'],
    performance['poor'])
rule2 = ctrl.Rule(experience['medium'] | success_rate['medium'],
    performance['average'])
rule3 = ctrl.Rule(experience['high'] & success_rate['high'],
    performance['excellent'])

performance_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])
performance_sim = ctrl.ControlSystemSimulation(performance_ctrl)

performance_sim.input['experience'] = 12
performance_sim.input['success_rate'] = 70

performance_sim.compute()

print(f"Predicted Performance Score: {performance_sim
    .output['performance']:.2f}")
```

Output

```
predicted performance score : 67.85
```