**53, 71) Automatic nought and crossess using random number**

import random

def print\_board(board):

for row in board:

print(" | ".join(row))

print("-" \* 9)

def check\_winner(board, player):

for row in board:

if all(cell == player for cell 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\_board\_full(board):

return all(cell != ' ' for row in board for cell in row)

def play\_game():

board = [[' ' for \_ in range(3)] for \_ in range(3)]

current\_player = 'X'

while True:

print\_board(board)

if current\_player == 'X':

print("Player X's turn:")

row, col = map(int, input("Enter row and column (e.g., 1 2): ").split())

else:

print("Player O's turn:")

row, col = random.randint(0, 2), random.randint(0, 2)

if 0 <= row < 3 and 0 <= col < 3 and board[row][col] == ' ':

board[row][col] = current\_player

if check\_winner(board, current\_player):

print\_board(board)

print(f"Player {current\_player} wins!")

break

elif is\_board\_full(board):

print\_board(board)

print("It's a tie!")

break

else:

current\_player = 'O' if current\_player == 'X' else 'X'

else:

print("Invalid move. Try again.")

if \_\_name\_\_ == "\_\_main\_\_":

play\_game()

**54, 69) CSP for sodoku game**

**Csp:** from constraint import Problem, AllDifferentConstraint

def create\_sudoku\_csp(board):

problem = Problem()

# Add variables for each cell

for i in range(9):

for j in range(9):

if board[i][j] == 0:

problem.addVariable(f"cell\_{i}\_{j}", range(1, 10))

# Add constraints for rows, columns, and 3x3 subgrids

for i in range(9):

# Row constraint

problem.addConstraint(AllDifferentConstraint(), [f"cell\_{i}\_{j}" for j in range(9)])

# Column constraint

problem.addConstraint(AllDifferentConstraint(), [f"cell\_{j}\_{i}" for j in range(9)])

# Subgrid constraint

for i in range(0, 9, 3):

for j in range(0, 9, 3):

problem.addConstraint(AllDifferentConstraint(), [f"cell\_{x}\_{y}" for x in range(i, i + 3) for y in range(j, j + 3)])

# Add initial assignments as constraints

for i in range(9):

for j in range(9):

if board[i][j] != 0:

problem.addConstraint(lambda value, i=i, j=j: value == board[i][j], (f"cell\_{i}\_{j}",))

return problem

def solve\_sudoku(board):

csp = create\_sudoku\_csp(board)

solution = csp.getSolution()

return solution

def print\_sudoku(board):

for row in board:

print(" ".join(map(str, row)))

if \_\_name\_\_ == "\_\_main\_\_":

# Example Sudoku board (0 represents empty cells)

sudoku\_board = [

[5, 3, 0, 0, 7, 0, 0, 0, 0],

[6, 0, 0, 1, 9, 5, 0, 0, 0],

[0, 9, 8, 0, 0, 0, 0, 6, 0],

[8, 0, 0, 0, 6, 0, 0, 0, 3],

[4, 0, 0, 8, 0, 3, 0, 0, 1],

[7, 0, 0, 0, 2, 0, 0, 0, 6],

[0, 6, 0, 0, 0, 0, 2, 8, 0],

[0, 0, 0, 4, 1, 9, 0, 0, 5],

[0, 0, 0, 0, 8, 0, 0, 7, 9]

]

print("Initial Sudoku Board:")

print\_sudoku(sudoku\_board)

solution = solve\_sudoku(sudoku\_board)

if solution:

print("\nSolved Sudoku:")

solved\_board = [[solution[f"cell\_{i}\_{j}"] for j in range(9)] for i in range(9)]

print\_sudoku(solved\_board)

else:

print("\nNo solution found.")

user input: from constraint import Problem, AllDifferentConstraint

def create\_sudoku\_csp(board):

problem = Problem()

# Add variables for each cell

for i in range(9):

for j in range(9):

if board[i][j] == 0:

problem.addVariable(f"cell\_{i}\_{j}", range(1, 10))

# Add constraints for rows, columns, and 3x3 subgrids

for i in range(9):

# Row constraint

problem.addConstraint(AllDifferentConstraint(), [f"cell\_{i}\_{j}" for j in range(9)])

# Column constraint

problem.addConstraint(AllDifferentConstraint(), [f"cell\_{j}\_{i}" for j in range(9)])

# Subgrid constraint

for i in range(0, 9, 3):

for j in range(0, 9, 3):

problem.addConstraint(AllDifferentConstraint(), [f"cell\_{x}\_{y}" for x in range(i, i + 3) for y in range(j, j + 3)])

# Add initial assignments as constraints

for i in range(9):

for j in range(9):

if board[i][j] != 0:

problem.addConstraint(lambda value, i=i, j=j: value == board[i][j], (f"cell\_{i}\_{j}",))

return problem

def solve\_sudoku(board):

csp = create\_sudoku\_csp(board)

solution = csp.getSolution()

return solution

def print\_sudoku(board):

for row in board:

print(" ".join(map(str, row)))

def get\_user\_input():

print("Enter the Sudoku board:")

board = []

for \_ in range(9):

row = list(map(int, input().split()))

board.append(row)

return board

if \_\_name\_\_ == "\_\_main\_\_":

sudoku\_board = get\_user\_input()

print("\nInitial Sudoku Board:")

print\_sudoku(sudoku\_board)

solution = solve\_sudoku(sudoku\_board)

if solution:

print("\nSolved Sudoku:")

solved\_board = [[solution[f"cell\_{i}\_{j}"] for j in range(9)] for i in range(9)]

print\_sudoku(solved\_board)

else:

print("\nNo solution found.")

**.**def is\_valid\_move(board, row, col, num):

# Check if 'num' is not in the current row, column, and subgrid

return (

all(num != board[row][i] for i in range(9)) and

all(num != board[i][col] for i in range(9)) and

all(num != board[row // 3 \* 3 + i][col // 3 \* 3 + j] for i in range(3) for j in range(3))

)

def solve\_sudoku(board):

empty\_cell = find\_empty\_cell(board)

if not empty\_cell:

# All cells are filled, the puzzle is solved

return True

row, col = empty\_cell

for num in range(1, 10):

if is\_valid\_move(board, row, col, num):

board[row][col] = num

if solve\_sudoku(board):

return True

# If placing 'num' at (row, col) doesn't lead to a solution, backtrack

board[row][col] = 0

# No valid number found, backtrack

return False

def find\_empty\_cell(board):

for i in range(9):

for j in range(9):

if board[i][j] == 0:

return i, j

return None

if \_\_name\_\_ == "\_\_main\_\_":

# Example Sudoku board (0 represents empty cells)

sudoku\_board = [

[5, 3, 0, 0, 7, 0, 0, 0, 0],

[6, 0, 0, 1, 9, 5, 0, 0, 0],

[0, 9, 8, 0, 0, 0, 0, 6, 0],

[8, 0, 0, 0, 6, 0, 0, 0, 3],

[4, 0, 0, 8, 0, 3, 0, 0, 1],

[7, 0, 0, 0, 2, 0, 0, 0, 6],

[0, 6, 0, 0, 0, 0, 2, 8, 0],

[0, 0, 0, 4, 1, 9, 0, 0, 5],

[0, 0, 0, 0, 8, 0, 0, 7, 9]

]

if solve\_sudoku(sudoku\_board):

print("Sudoku Solution:")

for row in sudoku\_board:

print(row)

else:

print("No solution found.")

user input: def is\_valid\_move(board, row, col, num):

return (

all(num != board[row][i] for i in range(9)) and

all(num != board[i][col] for i in range(9)) and

all(num != board[row // 3 \* 3 + i][col // 3 \* 3 + j] for i in range(3) for j in range(3))

)

def solve\_sudoku(board):

empty\_cell = find\_empty\_cell(board)

if not empty\_cell:

return True

row, col = empty\_cell

for num in range(1, 10):

if is\_valid\_move(board, row, col, num):

board[row][col] = num

if solve\_sudoku(board):

return True

board[row][col] = 0

return False

def find\_empty\_cell(board):

for i in range(9):

for j in range(9):

if board[i][j] == 0:

return i, j

return None

def print\_board(board):

for row in board:

print(" ".join(str(cell) if cell != 0 else '.' for cell in row))

def get\_user\_input():

board = []

print("Enter the Sudoku puzzle row by row. Use '0' for empty cells.")

for \_ in range(9):

row = input("Enter a row (9 digits separated by space): ").split()

board.append([int(cell) for cell in row])

return board

if \_\_name\_\_ == "\_\_main\_\_":

sudoku\_board = get\_user\_input()

print("\nInitial Sudoku Puzzle:")

print\_board(sudoku\_board)

if solve\_sudoku(sudoku\_board):

print("\nSudoku Solution:")

print\_board(sudoku\_board)

else:

print("No solution found.")

**50, 48, 68) Implement goal stack planning for block world problem**

class BlockWorld:

def \_\_init\_\_(self, initial\_state):

self.state = initial\_state

self.goal\_stack = []

def apply\_action(self, action):

if action[0] == 'move':

block, source, destination = action[1], action[2], action[3]

# Check if the block is in the state before removing it

if (block, source) in self.state:

self.state.remove((block, source))

self.state.append((block, destination))

else:

print(f"Warning: Block {block} not found at {source}. Skipping action.")

def is\_goal\_achieved(self):

return all(item in self.state for item in self.goal\_stack)

def execute\_plan(self, plan):

for action in plan:

self.apply\_action(action)

print(f"Action: {action}, State: {self.state}")

def block\_world\_planning(goal\_state, initial\_state):

block\_world = BlockWorld(initial\_state)

block\_world.goal\_stack = goal\_state

plan = []

for goal in goal\_state:

if goal not in block\_world.state:

sub\_plan = [('move', item[0], item[1], goal[1]) for item in block\_world.state if item[0] == goal[0]]

plan.extend(sub\_plan)

plan.append(('move', goal[0], goal[1], 'table'))

return plan

if \_\_name\_\_ == "\_\_main\_\_":

initial\_state = [('A', 'table'), ('B', 'A'), ('C', 'table')]

goal\_state = [('A', 'table'), ('C', 'A'), ('B', 'C')]

plan = block\_world\_planning(goal\_state, initial\_state)

block\_world = BlockWorld(initial\_state)

block\_world.execute\_plan(plan)

user input: class BlockWorld:

def \_\_init\_\_(self, initial\_state):

self.state = initial\_state

self.goal\_stack = []

def apply\_action(self, action):

if action[0] == 'move':

block, source, destination = action[1], action[2], action[3]

# Check if the block is in the state before removing it

if (block, source) in self.state:

self.state.remove((block, source))

self.state.append((block, destination))

else:

print(f"Warning: Block {block} not found at {source}. Skipping action.")

def is\_goal\_achieved(self):

return all(item in self.state for item in self.goal\_stack)

def execute\_plan(self, plan):

for action in plan:

self.apply\_action(action)

print(f"Action: {action}, State: {self.state}")

def get\_user\_input(prompt):

user\_input = input(prompt)

return [tuple(item.split()) for item in user\_input.split(',')]

if \_\_name\_\_ == "\_\_main\_\_":

print("Enter the initial state (e.g., A table, B A, C table):")

initial\_state = get\_user\_input("Initial state: ")

print("Enter the goal state (e.g., A table, C A, B C):")

goal\_state = get\_user\_input("Goal state: ")

plan = block\_world\_planning(goal\_state, initial\_state)

block\_world = BlockWorld(initial\_state)

print("\nInitial State:", block\_world.state)

print("Goal State:", goal\_state)

block\_world.execute\_plan(plan)

**67) write a program to implement hill climbing algorithm**

import random

def objective\_function(state):

# Define your objective function here

# For example, consider maximizing the sum of state values

return sum(state)

def generate\_neighbor(current\_state):

# Generate a neighboring state by randomly changing one element

neighbor = current\_state.copy()

index\_to\_change = random.randint(0, len(current\_state) - 1)

neighbor[index\_to\_change] += random.choice([-1, 1])

return neighbor

def hill\_climbing(initial\_state, max\_iterations):

current\_state = initial\_state

current\_value = objective\_function(current\_state)

for iteration in range(max\_iterations):

neighbor = generate\_neighbor(current\_state)

neighbor\_value = objective\_function(neighbor)

if neighbor\_value > current\_value:

current\_state = neighbor

current\_value = neighbor\_value

return current\_state, current\_value

if \_\_name\_\_ == "\_\_main\_\_":

# Define the problem - initial state and number of iterations

initial\_state = [random.randint(0, 10) for \_ in range(5)]

max\_iterations = 1000

# Run Hill Climbing algorithm

final\_state, final\_value = hill\_climbing(initial\_state, max\_iterations)

# Print results

print("Initial State:", initial\_state)

print("Final State:", final\_state)

print("Objective Value:", final\_value)

user input: import random

def objective\_function(state):

return sum(state)

def generate\_neighbor(current\_state):

neighbor = current\_state.copy()

index\_to\_change = random.randint(0, len(current\_state) - 1)

neighbor[index\_to\_change] += random.choice([-1, 1])

return neighbor

def hill\_climbing(initial\_state, max\_iterations):

current\_state = initial\_state

current\_value = objective\_function(current\_state)

for iteration in range(max\_iterations):

neighbor = generate\_neighbor(current\_state)

neighbor\_value = objective\_function(neighbor)

if neighbor\_value > current\_value:

current\_state = neighbor

current\_value = neighbor\_value

return current\_state, current\_value

def get\_user\_input():

length = int(input("Enter the length of the state vector: "))

initial\_state = [random.randint(0, 10) for \_ in range(length)]

max\_iterations = int(input("Enter the maximum number of iterations: "))

return initial\_state, max\_iterations

if \_\_name\_\_ == "\_\_main\_\_":

initial\_state, max\_iterations = get\_user\_input()

final\_state, final\_value = hill\_climbing(initial\_state, max\_iterations)

print("Initial State:", initial\_state)

print("Final State:", final\_state)

print("Objective Value:", final\_value)

**66) Build an expert system for traffic control signal**

class TrafficControlExpertSystem:

def \_\_init\_\_(self):

self.rules = [

{'condition': 'heavy\_traffic', 'signal': 'red'},

{'condition': 'moderate\_traffic', 'signal': 'yellow'},

{'condition': 'light\_traffic', 'signal': 'green'}

]

def get\_traffic\_signal(self, traffic\_condition):

for rule in self.rules:

if rule['condition'] == traffic\_condition:

return rule['signal']

return 'unknown'

if \_\_name\_\_ == "\_\_main\_\_":

expert\_system = TrafficControlExpertSystem()

# Example usage

traffic\_condition = input("Enter traffic condition (heavy\_traffic, moderate\_traffic, light\_traffic): ")

signal = expert\_system.get\_traffic\_signal(traffic\_condition)

if signal != 'unknown':

print(f"The recommended traffic signal is {signal}.")

else:

print("Unknown traffic condition.")

**62) Implement and compare different search algorithm such as A\*, depth-first search, on puzzle measure and compare the performance of the algorithm in terms of time complexity and solution quality**

import heapq

import time

class PuzzleNode:

def \_\_init\_\_(self, state, parent=None, action=None, cost=0):

self.state = state

self.parent = parent

self.action = action

self.cost = cost

self.heuristic = self.calculate\_heuristic()

def calculate\_heuristic(self):

# Implement a heuristic function (e.g., Manhattan distance)

pass

def \_\_lt\_\_(self, other):

return (self.cost + self.heuristic) < (other.cost + other.heuristic)

def a\_star\_search(initial\_state):

# Implement A\* algorithm

pass

class DFSNode:

def \_\_init\_\_(self, state, parent=None, action=None, cost=0):

self.state = state

self.parent = parent

self.action = action

self.cost = cost

def depth\_first\_search(initial\_state):

# Implement DFS algorithm

pass

def run\_algorithm(algorithm, initial\_state):

start\_time = time.time()

solution = algorithm(initial\_state)

end\_time = time.time()

return solution, end\_time - start\_time

if \_\_name\_\_ == "\_\_main\_\_":

# Define your puzzle problem and initial state

# For example, in an 8-puzzle problem, the initial state may look like:

initial\_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

# Run A\* algorithm

a\_star\_solution, a\_star\_time = run\_algorithm(a\_star\_search, initial\_state)

# Run DFS algorithm

dfs\_solution, dfs\_time = run\_algorithm(depth\_first\_search, initial\_state)

# Compare results

print("A\* Algorithm:")

print("Solution:", a\_star\_solution)

print("Time taken:", a\_star\_time)

print("\nDFS Algorithm:")

print("Solution:", dfs\_solution)

print("Time taken:", dfs\_time)

user input: import heapq

import time

class PuzzleNode:

def \_\_init\_\_(self, state, parent=None, action=None, cost=0):

self.state = state

self.parent = parent

self.action = action

self.cost = cost

self.heuristic = self.calculate\_heuristic()

def calculate\_heuristic(self):

# Implement a heuristic function (e.g., Manhattan distance)

pass

def \_\_lt\_\_(self, other):

return (self.cost + self.heuristic) < (other.cost + other.heuristic)

def a\_star\_search(initial\_state):

# Implement A\* algorithm

pass

class DFSNode:

def \_\_init\_\_(self, state, parent=None, action=None, cost=0):

self.state = state

self.parent = parent

self.action = action

self.cost = cost

def depth\_first\_search(initial\_state):

# Implement DFS algorithm

pass

def run\_algorithm(algorithm, initial\_state):

start\_time = time.time()

solution = algorithm(initial\_state)

end\_time = time.time()

return solution, end\_time - start\_time

def get\_user\_input():

print("Enter the initial state of the puzzle (3x3 matrix):")

initial\_state = []

for i in range(3):

row = list(map(int, input().split()))

initial\_state.append(row)

return initial\_state

if \_\_name\_\_ == "\_\_main\_\_":

# Get user input for the initial state

initial\_state = get\_user\_input()

# Run A\* algorithm

a\_star\_solution, a\_star\_time = run\_algorithm(a\_star\_search, initial\_state)

# Run DFS algorithm

dfs\_solution, dfs\_time = run\_algorithm(depth\_first\_search, initial\_state)

# Compare results

print("A\* Algorithm:")

print("Solution:", a\_star\_solution)

print("Time taken:", a\_star\_time)

print("\nDFS Algorithm:")

print("Solution:", dfs\_solution)

print("Time taken:", dfs\_time)

**47, 61) Constraint Satisfaction problem for crossword problem**

**Csp:** from constraint import Problem, AllDifferentConstraint

def create\_crossword\_csp(words, grid\_size):

problem = Problem()

# Define variables (word placements)

for word in words:

word\_length = len(word)

for i in range(grid\_size - word\_length + 1):

for j in range(grid\_size):

problem.addVariable(f"{word}\_{i}\_{j}", [0, 1])

# Add constraints for horizontal and vertical placement

for word1 in words:

for word2 in words:

if word1 != word2:

for i in range(grid\_size):

for j in range(grid\_size):

if i + len(word1) <= grid\_size and j + len(word2) <= grid\_size:

# Check horizontal placement

constraint\_expr = [(f"{word1}\_{i}\_{k}", 1) for k in range(len(word1))] + \

[(f"{word2}\_{j}\_{k}", 1) for k in range(len(word2))]

problem.addConstraint(AllDifferentConstraint(), constraint\_expr)

# Check vertical placement

constraint\_expr = [(f"{word1}\_{k}\_{j}", 1) for k in range(len(word1))] + \

[(f"{word2}\_{k}\_{i}", 1) for k in range(len(word2))]

problem.addConstraint(AllDifferentConstraint(), constraint\_expr)

return problem

def solve\_crossword(words, grid\_size):

csp = create\_crossword\_csp(words, grid\_size)

solution = csp.getSolution()

return solution

def print\_crossword\_solution(solution, words, grid\_size):

grid = [[' ' for \_ in range(grid\_size)] for \_ in range(grid\_size)]

for var, value in solution.items():

if value == 1:

word, i, j = var.split('\_')

i, j = int(i), int(j)

for k, letter in enumerate(word):

if word[i] != grid[i + k][j]:

grid[i + k][j] = letter

# Print the crossword grid

for row in grid:

print(' '.join(row))

if \_\_name\_\_ == "\_\_main\_\_":

# Example words for the crossword

crossword\_words = ["python", "java", "swift", "csharp", "ruby", "html"]

# Size of the crossword grid (assumes a square grid)

crossword\_grid\_size = 10

# Solve the crossword puzzle

crossword\_solution = solve\_crossword(crossword\_words, crossword\_grid\_size)

if crossword\_solution:

print("\nCrossword Solution:")

print\_crossword\_solution(crossword\_solution, crossword\_words, crossword\_grid\_size)

else:

print("\nNo solution found.")

user input: from constraint import Problem, AllDifferentConstraint

def create\_crossword\_csp(words, grid\_size):

problem = Problem()

# Define variables (word placements)

for word in words:

word\_length = len(word)

for i in range(grid\_size - word\_length + 1):

for j in range(grid\_size):

problem.addVariable(f"{word}\_{i}\_{j}", [0, 1])

# Add constraints for horizontal and vertical placement

for word1 in words:

for word2 in words:

if word1 != word2:

for i in range(grid\_size):

for j in range(grid\_size):

if i + len(word1) <= grid\_size and j + len(word2) <= grid\_size:

# Check horizontal placement

constraint\_expr = [(f"{word1}\_{i}\_{k}", 1) for k in range(len(word1))] + \

[(f"{word2}\_{j}\_{k}", 1) for k in range(len(word2))]

problem.addConstraint(AllDifferentConstraint(), constraint\_expr)

# Check vertical placement

constraint\_expr = [(f"{word1}\_{k}\_{j}", 1) for k in range(len(word1))] + \

[(f"{word2}\_{k}\_{i}", 1) for k in range(len(word2))]

problem.addConstraint(AllDifferentConstraint(), constraint\_expr)

return problem

def solve\_crossword(words, grid\_size):

csp = create\_crossword\_csp(words, grid\_size)

solution = csp.getSolution()

return solution

def print\_crossword\_solution(solution, words, grid\_size):

grid = [[' ' for \_ in range(grid\_size)] for \_ in range(grid\_size)]

for var, value in solution.items():

if value == 1:

word, i, j = var.split('\_')

i, j = int(i), int(j)

for k, letter in enumerate(word):

if word[i] != grid[i + k][j]:

grid[i + k][j] = letter

# Print the crossword grid

for row in grid:

print(' '.join(row))

def get\_user\_input():

print("Enter the words for the crossword (separated by spaces):")

words = input().split()

print("Enter the size of the crossword grid:")

grid\_size = int(input())

return words, grid\_size

if \_\_name\_\_ == "\_\_main\_\_":

crossword\_words, crossword\_grid\_size = get\_user\_input()

# Solve the crossword puzzle

crossword\_solution = solve\_crossword(crossword\_words, crossword\_grid\_size)

if crossword\_solution:

print("\nCrossword Solution:")

print\_crossword\_solution(crossword\_solution, crossword\_words, crossword\_grid\_size)

else:

print("\nNo solution found.")

2. def crossword\_csp(words, grid):

assignment = {}

solutions = backtrack(assignment, words, grid)

return solutions if solutions else []

def backtrack(assignment, words, grid):

if len(assignment) == len(grid) \* len(grid[0]):

return [assignment.copy()]

i, j = divmod(len(assignment), len(grid[0]))

if grid[i][j] == '.':

for word in words:

new\_assignment = assignment.copy()

new\_assignment[f"({i},{j})"] = word

result = backtrack(new\_assignment, words, grid)

if result:

return result

return None

if \_\_name\_\_ == "\_\_main\_\_":

crossword\_grid = [

['.', '.', '.', '.', '.', '.'],

['.', '.', '.', '.', '.', '.'],

['.', '.', '.', '.', '.', '.'],

['.', '.', '.', '.', '.', '.'],

['.', '.', '.', '.', '.', '.'],

['.', '.', '.', '.', '.', '.'],

]

word\_list = ["python", "java", "swift", "csharp", "ruby", "html"]

solutions = crossword\_csp(word\_list, crossword\_grid)

for solution in solutions:

for i in range(len(crossword\_grid)):

for j in range(len(crossword\_grid[i])):

print(solution.get(f"({i},{j})", '.'), end=' ')

print()

print("\n")

user input: def crossword\_csp(words, grid):

assignment = {}

solutions = backtrack(assignment, words, grid)

return solutions if solutions else []

def backtrack(assignment, words, grid):

if len(assignment) == len(grid) \* len(grid[0]):

return [assignment.copy()]

i, j = divmod(len(assignment), len(grid[0]))

if grid[i][j] == '.':

for word in words:

new\_assignment = assignment.copy()

new\_assignment[f"({i},{j})"] = word

result = backtrack(new\_assignment, words, grid)

if result:

return result

return None

def get\_user\_input():

print("Enter the crossword grid (use '.' for empty cells):")

crossword\_grid = []

for \_ in range(6): # Assuming a 6x6 grid for simplicity

row = list(input())

crossword\_grid.append(row)

return crossword\_grid

if \_\_name\_\_ == "\_\_main\_\_":

crossword\_grid = get\_user\_input()

print("Enter the word list:")

word\_list = input().split()

solutions = crossword\_csp(word\_list, crossword\_grid)

for solution in solutions:

for i in range(len(crossword\_grid)):

for j in range(len(crossword\_grid[i])):

print(solution.get(f"({i},{j})", '.'), end=' ')

print()

print("\n")

**43,60) Build an expert system for financial planning**

class FinancialExpertSystem:

def \_\_init\_\_(self):

self.knowledge\_base = {}

def ask\_question(self, question):

response = input(question + " (yes/no): ").lower()

return response == 'yes'

def get\_numeric\_input(self, prompt):

while True:

try:

return float(input(prompt))

except ValueError:

print("Please enter a valid numeric value.")

def recommend\_investment\_strategy(self):

age = int(input("What is your age? "))

income = self.get\_numeric\_input("What is your annual income? $")

debt = self.get\_numeric\_input("Do you have any outstanding debt? $")

risk\_tolerance = self.ask\_question("Are you comfortable with high-risk investments?")

# Basic recommendation logic

recommendation = "Conservative" if age > 50 or debt > 10000 else "Moderate"

if risk\_tolerance:

recommendation = "Aggressive"

self.knowledge\_base['age'] = age

self.knowledge\_base['income'] = income

self.knowledge\_base['debt'] = debt

self.knowledge\_base['risk\_tolerance'] = risk\_tolerance

return recommendation

if \_\_name\_\_ == "\_\_main\_\_":

financial\_expert = FinancialExpertSystem()

print("Welcome to the Financial Planning Expert System!")

investment\_strategy = financial\_expert.recommend\_investment\_strategy()

print("\nBased on your responses, we recommend an", investment\_strategy, "investment strategy.")

print("Thank you for using the Financial Planning Expert System!")

**51) Develop a conversational AI chatbot in the field of food ordering system**

import random

class FoodOrderingChatbot:

def \_\_init\_\_(self):

self.menu = {

'1': {'name': 'Pizza', 'price': 10.99},

'2': {'name': 'Burger', 'price': 5.99},

'3': {'name': 'Pasta', 'price': 8.99},

}

self.user\_order = {}

def welcome(self):

return "Welcome to the Food Ordering Chatbot! How can I help you today?"

def display\_menu(self):

menu\_text = "Menu:\n"

for item\_id, details in self.menu.items():

menu\_text += f"{item\_id}. {details['name']} - ${details['price']}\n"

return menu\_text

def order\_food(self, food\_item\_id):

if food\_item\_id in self.menu:

item = self.menu[food\_item\_id]

self.user\_order = {'name': item['name'], 'price': item['price']}

return f"You have ordered {item['name']}. Your total is ${item['price']}. Would you like to place the order?"

else:

return "Sorry, I couldn't find that item in the menu. Please try again."

def check\_order\_status(self):

if self.user\_order:

return f"Your current order status is: {self.user\_order['name']} is in process."

else:

return "You don't have any active orders. Would you like to place a new order?"

def handle\_input(self, user\_input):

user\_input = user\_input.lower()

if "order" in user\_input:

food\_item\_id = input("Enter the item number you want to order: ")

return self.order\_food(food\_item\_id)

elif "status" in user\_input:

return self.check\_order\_status()

elif "menu" in user\_input:

return self.display\_menu()

elif "exit" in user\_input or "quit" in user\_input:

return "Thank you for using the Food Ordering Chatbot. Goodbye!"

else:

return "Sorry, I didn't understand that. Can you please repeat?"

def run(self):

print(self.welcome())

while True:

user\_input = input("User: ")

response = self.handle\_input(user\_input)

print(f"Bot: {response}")

if "exit" in user\_input or "quit" in user\_input:

break

if \_\_name\_\_ == "\_\_main\_\_":

chatbot = FoodOrderingChatbot()

chatbot.run()

**49) Use heuristic search techniques to implement hill climbing algorithm**

import random

def objective\_function(x):

# Example objective function (you can replace this with your own function)

return -((x - 5) \*\* 2) # Maximizing a quadratic function

def hill\_climbing(initial\_solution, max\_iterations):

current\_solution = initial\_solution

for \_ in range(max\_iterations):

neighbors = generate\_neighbors(current\_solution)

neighbor\_values = [objective\_function(neighbor) for neighbor in neighbors]

best\_neighbor = neighbors[neighbor\_values.index(max(neighbor\_values))]

if objective\_function(best\_neighbor) > objective\_function(current\_solution):

current\_solution = best\_neighbor

return current\_solution

def generate\_neighbors(solution):

# Simple function to generate neighbors by adding or subtracting a small random value

return [solution + random.uniform(-0.5, 0.5) for \_ in range(5)]

if \_\_name\_\_ == "\_\_main\_\_":

# Example usage

initial\_solution = random.uniform(0, 10)

max\_iterations = 100

best\_solution = hill\_climbing(initial\_solution, max\_iterations)

print(f"Best solution found: {best\_solution}")

print(f"Objective function value: {objective\_function(best\_solution)}")

user input: import random

def objective\_function(x):

# Example objective function (you can replace this with your own function)

return -((x - 5) \*\* 2) # Maximizing a quadratic function

def hill\_climbing(initial\_solution, max\_iterations):

current\_solution = initial\_solution

for \_ in range(max\_iterations):

neighbors = generate\_neighbors(current\_solution)

neighbor\_values = [objective\_function(neighbor) for neighbor in neighbors]

best\_neighbor = neighbors[neighbor\_values.index(max(neighbor\_values))]

if objective\_function(best\_neighbor) > objective\_function(current\_solution):

current\_solution = best\_neighbor

return current\_solution

def generate\_neighbors(solution):

# Simple function to generate neighbors by adding or subtracting a small random value

return [solution + random.uniform(-0.5, 0.5) for \_ in range(5)]

if \_\_name\_\_ == "\_\_main\_\_":

# User input for initial solution and number of iterations

initial\_solution = float(input("Enter the initial solution: "))

max\_iterations = int(input("Enter the number of iterations: "))

best\_solution = hill\_climbing(initial\_solution, max\_iterations)

print(f"Best solution found: {best\_solution}")

print(f"Objective function value: {objective\_function(best\_solution)}")

**46) CSP for n queen problem**

def is\_safe(board, row, col, n):

# Check if there is a queen in the same row

for i in range(col):

if board[row][i] == 1:

return False

# Check upper diagonal on the left side

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

# Check lower diagonal on the left side

for i, j in zip(range(row, n, 1), range(col, -1, -1)):

if board[i][j] == 1:

return False

return True

def solve\_n\_queens\_util(board, col, n):

if col == n:

return True

for i in range(n):

if is\_safe(board, i, col, n):

board[i][col] = 1

if solve\_n\_queens\_util(board, col + 1, n):

return True

board[i][col] = 0

return False

def solve\_n\_queens(n):

board = [[0 for \_ in range(n)] for \_ in range(n)]

if not solve\_n\_queens\_util(board, 0, n):

print("Solution does not exist.")

return

print\_solution(board)

def print\_solution(board):

for row in board:

print(" ".join(["Q" if col == 1 else "." for col in row]))

if \_\_name\_\_ == "\_\_main\_\_":

n = int(input("Enter the number of queens: "))

solve\_n\_queens(n)

**csp:**

pip install python-constraint

from constraint import Problem, AllDifferentConstraint

def n\_queens(n):

problem = Problem()

# Define variables (queen positions)

problem.addVariables(range(n), range(n))

# Add constraints

problem.addConstraint(AllDifferentConstraint())

def diagonal\_constraint(q1, q2, d):

return abs(q1 - q2) != d

for i in range(n):

for j in range(i + 1, n):

problem.addConstraint(diagonal\_constraint, (i, j, j - i))

problem.addConstraint(diagonal\_constraint, (i, j, i - j))

# Find solution

solutions = problem.getSolutions()

return solutions

def print\_solution(solution):

for row in solution:

line = ["Q" if col == row else "." for col in solution[row]]

print(" ".join(line))

if \_\_name\_\_ == "\_\_main\_\_":

n = int(input("Enter the number of queens: "))

solutions = n\_queens(n)

print(f"\nTotal solutions found: {len(solutions)}")

if solutions:

print("\nOne of the solutions:")

print\_solution(solutions[0])