1. Write automatic program for 0and X

import random

def print\_board(board):

for row in board:

print(" | ".join(row))

print("-" \* 5)

def is\_winner(board, player):

# Check rows, columns, and diagonals for a win

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

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):

# Check if the board is full

return all(board[i][j] != ' ' for i in range(3) for j in range(3))

def get\_available\_moves(board):

# Get a list of available moves (empty cells)

return [(i, j) for i in range(3) for j in range(3) if board[i][j] == ' ']

def make\_move(board, player, move):

# Make a move on the board

i, j = move

board[i][j] = player

def get\_user\_move():

# Get user's move

while True:

try:

move = tuple(map(int, input("Enter your move (row and column, separated by space): ").split()))

if 0 <= move[0] < 3 and 0 <= move[1] < 3:

return move

else:

print("Invalid move. Try again.")

except ValueError:

print("Invalid input. Try again.")

def main():

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

user = 'X'

computer = 'O'

while True:

print\_board(board)

# User's move

user\_move = get\_user\_move()

make\_move(board, user, user\_move)

# Check if user wins

if is\_winner(board, user):

print\_board(board)

print("Congratulations! You win!")

break

# Check if the board is full

if is\_board\_full(board):

print\_board(board)

print("It's a tie!")

break

# Computer's move

computer\_move = random.choice(get\_available\_moves(board))

make\_move(board, computer, computer\_move)

# Check if computer wins

if is\_winner(board, computer):

print\_board(board)

print("Sorry, you lose. Try again!")

break

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

main()

2.use heuristic search techniques to implement hill-climbing algorithm

3. constraint satisfaction problem for sudoku game

def print\_board(board):

for row in board:

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

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

# Check if the move is valid in the current row and column

if num in board[row] or num in [board[i][col] for i in range(9)]:

return False

# Check if the move is valid in the 3x3 subgrid

start\_row, start\_col = 3 \* (row // 3), 3 \* (col // 3)

for i in range(start\_row, start\_row + 3):

for j in range(start\_col, start\_col + 3):

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

return False

return True

def find\_empty\_location(board):

# Find the first empty location (cell with 0)

for i in range(9):

for j in range(9):

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

return i, j

return None, None # No empty location found, puzzle is solved

def solve\_sudoku(board):

row, col = find\_empty\_location(board)

# If there is no empty location, the puzzle is solved

if row is None:

return True

# Try placing digits 1 through 9 in the empty location

for num in range(1, 10):

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

board[row][col] = num

# Recursively try to solve the remaining puzzle

if solve\_sudoku(board):

return True

# If placing num in the current location doesn't lead to a solution, backtrack

board[row][col] = 0

# No valid move, trigger backtracking

return False

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

# Example Sudoku puzzle (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("Sudoku Puzzle:")

print\_board(sudoku\_board)

print("\nSolving...\n")

if solve\_sudoku(sudoku\_board):

print("Sudoku Solution:")

print\_board(sudoku\_board)

else:

print("No solution exists.")

2\*2

def print\_board(board):

for row in board:

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

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

# Check if the move is valid in the current row and column

if num in board[row] or num in [board[i][col] for i in range(4)]:

return False

# Check if the move is valid in the 2x2 subgrid

start\_row, start\_col = 2 \* (row // 2), 2 \* (col // 2)

for i in range(start\_row, start\_row + 2):

for j in range(start\_col, start\_col + 2):

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

return False

return True

def find\_empty\_location(board):

# Find the first empty location (cell with 0)

for i in range(4):

for j in range(4):

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

return i, j

return None, None # No empty location found, puzzle is solved

def solve\_sudoku(board):

row, col = find\_empty\_location(board)

# If there is no empty location, the puzzle is solved

if row is None:

return True

# Try placing digits 1 through 4 in the empty location

for num in range(1, 5):

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

board[row][col] = num

# Recursively try to solve the remaining puzzle

if solve\_sudoku(board):

return True

# If placing num in the current location doesn't lead to a solution, backtrack

board[row][col] = 0

# No valid move, trigger backtracking

return False

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

# Example 4x4 Sudoku puzzle (0 represents empty cells)

sudoku\_board = [

[0, 0, 2, 1],

[0, 1, 0, 0],

[0, 0, 0, 0],

[0, 2, 0, 0]

]

print("Sudoku Puzzle:")

print\_board(sudoku\_board)

print("\nSolving...\n")

if solve\_sudoku(sudoku\_board):

print("Sudoku Solution:")

print\_board(sudoku\_board)

else:

print("No solution exists.")

4.write a program to implement hill climbing algorithm

5.implement and compare different search algorithm such as A\*,bffs,dfs on puzzle measure and compare the performance of algorithm in terms of time complexity and solution quality

from queue import Queue, LifoQueue

import heapq

import time

import random

class PuzzleState:

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

self.state = state

self.parent = parent

self.action = action

self.cost = 0

def get\_blank\_position(state):

for i in range(len(state)):

for j in range(len(state[i])):

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

return i, j

def actions(state):

i, j = get\_blank\_position(state)

possible\_actions = []

if i > 0:

possible\_actions.append("UP")

if i < len(state) - 1:

possible\_actions.append("DOWN")

if j > 0:

possible\_actions.append("LEFT")

if j < len(state[0]) - 1:

possible\_actions.append("RIGHT")

return possible\_actions

def result(state, action):

i, j = get\_blank\_position(state)

if action == "UP":

state[i][j], state[i - 1][j] = state[i - 1][j], state[i][j]

elif action == "DOWN":

state[i][j], state[i + 1][j] = state[i + 1][j], state[i][j]

elif action == "LEFT":

state[i][j], state[i][j - 1] = state[i][j - 1], state[i][j]

elif action == "RIGHT":

state[i][j], state[i][j + 1] = state[i][j + 1], state[i][j]

def print\_state(state):

for row in state:

print(row)

print()

def is\_goal(state, goal\_state):

return state == goal\_state

def bfs(start\_state, goal\_state):

start\_time = time.time()

explored = set()

frontier = Queue()

initial\_node = PuzzleState(start\_state)

frontier.put(initial\_node)

while not frontier.empty():

current\_node = frontier.get()

current\_state = current\_node.state

if is\_goal(current\_state, goal\_state):

end\_time = time.time()

return current\_node, end\_time - start\_time

explored.add(tuple(map(tuple, current\_state)))

for action in actions(current\_state):

child\_state = [row.copy() for row in current\_state]

result(child\_state, action)

if tuple(map(tuple, child\_state)) not in explored:

child\_node = PuzzleState(child\_state, current\_node, action)

frontier.put(child\_node)

return None, None

def dfs(start\_state, goal\_state):

start\_time = time.time()

explored = set()

frontier = LifoQueue()

initial\_node = PuzzleState(start\_state)

frontier.put(initial\_node)

while not frontier.empty():

current\_node = frontier.get()

current\_state = current\_node.state

if is\_goal(current\_state, goal\_state):

end\_time = time.time()

return current\_node, end\_time - start\_time

explored.add(tuple(map(tuple, current\_state)))

for action in actions(current\_state):

child\_state = [row.copy() for row in current\_state]

result(child\_state, action)

if tuple(map(tuple, child\_state)) not in explored:

child\_node = PuzzleState(child\_state, current\_node, action)

frontier.put(child\_node)

return None, None

def heuristic(state, goal\_state):

# Manhattan distance heuristic

total\_distance = 0

for i in range(len(state)):

for j in range(len(state[i])):

value = state[i][j]

if value != 0:

goal\_i, goal\_j = divmod(value - 1, len(state[i]))

total\_distance += abs(i - goal\_i) + abs(j - goal\_j)

return total\_distance

def astar(start\_state, goal\_state):

start\_time = time.time()

explored = set()

frontier = []

heapq.heappush(frontier, (0, PuzzleState(start\_state)))

cost\_so\_far = {tuple(map(tuple, start\_state)): 0}

while frontier:

\_, current\_node = heapq.heappop(frontier)

current\_state = current\_node.state

if is\_goal(current\_state, goal\_state):

end\_time = time.time()

return current\_node, end\_time - start\_time

explored.add(tuple(map(tuple, current\_state)))

for action in actions(current\_state):

child\_state = [row.copy() for row in current\_state]

result(child\_state, action)

child\_cost = current\_node.cost + 1 + heuristic(child\_state, goal\_state)

if tuple(map(tuple, child\_state)) not in explored or child\_cost < cost\_so\_far.get(tuple(map(tuple, child\_state)), float('inf')):

child\_node = PuzzleState(child\_state, current\_node, action)

child\_node.cost = current\_node.cost + 1

cost\_so\_far[tuple(map(tuple, child\_state))] = child\_cost

heapq.heappush(frontier, (child\_cost, child\_node))

return None, None

def generate\_puzzle(size):

numbers = list(range(size \* size))

random.shuffle(numbers)

puzzle = [numbers[i:i+size] for i in range(0, size\*size, size)]

return puzzle

def main():

size = 3

start\_state = generate\_puzzle(size)

goal\_state = generate\_puzzle(size)

print("Initial State:")

print\_state(start\_state)

print("Goal State:")

print\_state(goal\_state)

print("\nBFS:")

bfs\_solution, bfs\_time = bfs(start\_state, goal\_state)

if bfs\_solution:

print("Solution:")

while bfs\_solution:

print\_state(bfs\_solution.state)

bfs\_solution = bfs\_solution.parent

print("Time:", bfs\_time)

else:

print("No solution found.")

print("\nDFS:")

dfs\_solution, dfs\_time = dfs(start\_state, goal\_state)

if dfs\_solution:

print("Solution:")

while dfs\_solution:

print\_state(dfs\_solution.state)

dfs\_solution = dfs\_solution.parent

print("Time:", dfs\_time)

else:

print("No solution found.")

print("\nA\*:")

astar\_solution, astar\_time = astar(start\_state, goal\_state)

if astar\_solution:

print("Solution:")

while astar\_solution:

print\_state(astar\_solution.state)

astar\_solution = astar\_solution.parent

print("Time:", astar\_time)

else:

print("No solution found.")

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

main()

6.implement goal stack planning for block world problem

7.develop a chatbot in field of medical system

8.expert system for financial planning

class FinancialExpertSystem:

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

self.income = 0

self.expenses = 0

self.savings = 0

def get\_user\_input(self):

try:

self.income = float(input("Enter your monthly income: "))

self.expenses = float(input("Enter your monthly expenses: "))

except ValueError:

print("Invalid input. Please enter valid numbers.")

self.get\_user\_input()

def calculate\_savings(self):

self.savings = self.income - self.expenses

def analyze\_situation(self):

if self.savings < 0:

print("You are spending more than you earn. Consider reducing expenses.")

elif 0 <= self.savings < self.income \* 0.1:

print("Your savings are below 10% of your income. Consider saving more.")

elif self.income \* 0.1 <= self.savings < self.income \* 0.5:

print("Your savings are within a healthy range. Keep it up!")

else:

print("Excellent job! Your savings are more than 50% of your income.")

def main():

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

financial\_expert = FinancialExpertSystem()

financial\_expert.get\_user\_input()

financial\_expert.calculate\_savings()

financial\_expert.analyze\_situation()

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

main()

9.csp for nqueen problem

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):

# Check for queens in the same column

for i in range(row):

if board[i][col]:

return False

# Check for queens in the left upper diagonal

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

if board[i][j]:

return False

# Check for queens in the left lower diagonal

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

if board[i][j]:

return False

return True

def solve\_n\_queens(board, row):

if row == len(board):

# All queens are placed, print the solution

print\_solution(board)

return

for col in range(len(board)):

if is\_safe(board, row, col):

# Place queen and move to the next row

board[row][col] = True

solve\_n\_queens(board, row + 1)

# Backtrack and try the next column

board[row][col] = False

def n\_queens(n):

# Initialize the chessboard

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

solve\_n\_queens(board, 0)

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

n\_queens(4)

10.use heuristic search technique to implement hill climbing algorithm

import random

def initial\_state(n):

"""Generate a random initial state for the N-Queens problem."""

return [random.randint(0, n-1) for \_ in range(n)]

def attacking\_pairs(state):

"""Calculate the number of attacking pairs in the given state."""

n = len(state)

pairs = 0

for i in range(n):

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

if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):

pairs += 1

return pairs

def hill\_climbing(n, max\_iter=1000):

"""Solve the N-Queens problem using Hill Climbing."""

current\_state = initial\_state(n)

for \_ in range(max\_iter):

if attacking\_pairs(current\_state) == 0:

# Goal state found

return current\_state

neighbors = []

for col in range(n):

for row in range(n):

if current\_state[col] != row:

neighbor = list(current\_state)

neighbor[col] = row

neighbors.append(neighbor)

neighbors.sort(key=attacking\_pairs)

if attacking\_pairs(neighbors[0]) >= attacking\_pairs(current\_state):

# No better neighbor found

break

current\_state = neighbors[0]

return current\_state

def print\_board(state):

"""Print the chessboard."""

n = len(state)

for row in range(n):

line = ""

for col in range(n):

line += "Q" if state[col] == row else "."

print(line)

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

n = 8

final\_state = hill\_climbing(n)

print("Final State:")

print\_board(final\_state)

11.csp for cossword problem

1

from constraint import Problem, AllDifferentConstraint

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

problem = Problem()

# Variables: one for each cell in the grid

variables = [(i, j) for i in range(grid\_size) for j in range(grid\_size)]

for variable in variables:

problem.addVariable(variable, words)

# Constraints: words in rows and columns should be different

for i in range(grid\_size):

row\_variables = [(i, j) for j in range(grid\_size)]

col\_variables = [(j, i) for j in range(grid\_size)]

problem.addConstraint(AllDifferentConstraint(), row\_variables)

problem.addConstraint(AllDifferentConstraint(), col\_variables)

return problem

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

for i in range(grid\_size):

row = [solution[(i, j)] for j in range(grid\_size)]

print(" ".join(row))

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

grid\_size = 4

words = ["cat", "dog", "bat", "rat"]

crossword\_problem = create\_crossword\_csp(grid\_size, words)

solutions = crossword\_problem.getSolutions()

if solutions:

print("One possible crossword solution:")

print\_solution(solutions[0], grid\_size)

else:

print("No solution found.")

pip install python-constraint

2

def is\_valid\_placement(grid, word, position, direction):

row, col = position

word\_len = len(word)

if direction == "across":

if col + word\_len > len(grid[0]):

return False

for i in range(word\_len):

if grid[row][col + i] != '.' and grid[row][col + i] != word[i]:

return False

elif direction == "down":

if row + word\_len > len(grid):

return False

for i in range(word\_len):

if grid[row + i][col] != '.' and grid[row + i][col] != word[i]:

return False

return True

def place\_word(grid, word, position, direction):

row, col = position

if direction == "across":

for i in range(len(word)):

grid[row][col + i] = word[i]

elif direction == "down":

for i in range(len(word)):

grid[row + i][col] = word[i]

def crossword\_puzzle(words):

grid\_size = 4

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

for word in words:

placed = False

while not placed:

direction = "across" if len(word) % 2 == 0 else "down"

position = (random.randint(0, grid\_size - 1), random.randint(0, grid\_size - 1))

if is\_valid\_placement(grid, word, position, direction):

place\_word(grid, word, position, direction)

placed = True

return grid

def print\_crossword(grid):

for row in grid:

print(" ".join(row))

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

import random

words = ["cat", "dog", "bat", "rat"]

crossword\_grid = crossword\_puzzle(words)

print("Crossword Puzzle:")

print\_crossword(crossword\_grid)

12.develop ai chat bot in field of food orderings system

import random

class FoodOrderingChatbot:

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

self.menu = {

"Burger": 5.99,

"Pizza": 8.99,

"Salad": 4.99,

"Drink": 1.99

}

self.cart = []

def display\_menu(self):

print("Menu:")

for item, price in self.menu.items():

print(f"{item}: ${price}")

def add\_to\_cart(self, item):

if item in self.menu:

self.cart.append((item, self.menu[item]))

print(f"{item} added to your cart.")

else:

print("Sorry, we don't have that item on the menu.")

def view\_cart(self):

if not self.cart:

print("Your cart is empty.")

else:

print("Your Cart:")

for item, price in self.cart:

print(f"{item}: ${price}")

def place\_order(self):

if not self.cart:

print("Your cart is empty. Please add items before placing an order.")

else:

total\_price = sum(price for \_, price in self.cart)

print(f"Total Price: ${total\_price}")

print("Thank you for placing your order! Your food will be delivered soon.")

self.cart = [] # Clear the cart after placing the order

def chat(self):

print("Welcome to Food Ordering Chatbot!")

while True:

user\_input = input("You: ").lower()

if "menu" in user\_input:

self.display\_menu()

elif "add" in user\_input:

item = user\_input.split("add")[1].strip()

self.add\_to\_cart(item)

elif "cart" in user\_input:

self.view\_cart()

elif "order" in user\_input:

self.place\_order()

elif "exit" in user\_input:

print("Thank you for using Food Ordering Chatbot. Goodbye!")

break

else:

responses = ["I'm sorry, I didn't understand that.", "Can you please rephrase?", "I'm just a simple chatbot for food orders."]

print(random.choice(responses))

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

chatbot = FoodOrderingChatbot()

chatbot.chat()

13.build an expert system for traffic control system

class TrafficControlExpertSystem:

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

self.weather\_conditions = ["clear", "rainy", "snowy"]

self.traffic\_conditions = ["light", "moderate", "heavy"]

def get\_traffic\_signal\_state(self, weather, traffic):

if weather not in self.weather\_conditions or traffic not in self.traffic\_conditions:

return "Invalid input. Please provide valid weather and traffic conditions."

if weather == "clear":

if traffic == "light":

return "Green signal"

elif traffic == "moderate":

return "Yellow signal"

elif traffic == "heavy":

return "Red signal"

elif weather == "rainy":

if traffic == "light":

return "Yellow signal"

elif traffic == "moderate" or traffic == "heavy":

return "Red signal"

elif weather == "snowy":

if traffic == "light" or traffic == "moderate":

return "Yellow signal"

elif traffic == "heavy":

return "Red signal"

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

traffic\_system = TrafficControlExpertSystem()

# Example usage

weather\_condition = input("Enter weather condition (clear, rainy, snowy): ").lower()

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

signal\_state = traffic\_system.get\_traffic\_signal\_state(weather\_condition, traffic\_condition)

print(f"Traffic Signal State: {signal\_state}")

13.chatbot for education system

import random

class EducationChatbot:

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

self.courses = {

"math": "Mathematics",

"sci": "Science",

"eng": "English",

"hist": "History",

"comp": "Computer Science"

}

self.schedule = {

"mon": {"math": "8:00 AM", "sci": "9:30 AM", "eng": "11:00 AM"},

"tue": {"hist": "8:30 AM", "comp": "10:00 AM"}

# Add more days and courses as needed

}

def get\_course\_info(self, course\_code):

if course\_code in self.courses:

return f"{self.courses[course\_code]} is a great course!"

else:

return "Sorry, I don't have information about that course."

def get\_schedule(self, day):

if day in self.schedule:

return self.schedule[day]

else:

return "Sorry, I don't have a schedule for that day."

def chat(self):

print("Welcome to the Education Chatbot!")

while True:

user\_input = input("You: ").lower()

if "course" in user\_input:

course\_code = input("Enter course code: ").lower()

response = self.get\_course\_info(course\_code)

elif "schedule" in user\_input:

day = input("Enter day (mon, tue, etc.): ").lower()

response = self.get\_schedule(day)

elif "exit" in user\_input:

print("Thank you for using the Education Chatbot. Goodbye!")

break

else:

responses = ["I'm sorry, I didn't understand that.", "Can you please rephrase?", "I'm just a simple chatbot for education."]

response = random.choice(responses)

print(response)

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

chatbot = EducationChatbot()

chatbot.chat()