Name: Mozeb Ahmed Khan

Roll No: 20F-0161

Section: BCS-6A

Course: Artificial Intelligence

Assignment No: 3

Problem 1 [Local Search Algorithms]

Code:

```
import random
import math

def generateRandomStartState(numOfQueens):
    """
        Generates a random starting state for the n-queens problem.

        Parameters:
            - numOfQueens (int): The number of queens to place on the
board.

        Returns:
            - startState (list): A list representing the initial state of
the board.
              Each element of the list represents the row position of a
queen on the board.
        """
    startState = []
    for row in range(numOfQueens):
        startState.append(random.randint(0, numOfQueens - 1))
    return startState

def generateSuccessorStates(state):
    """
        Generates all possible successor states for the given state.
```

```
        Parameters:
            - state (list): A list representing the current state of the
board.

        Returns:
            - successors (list): A list of all possible successor states to
the current state.
        """
    n = len(state)
    successors = []
    for col in range(n):
        for row in range(n):
            if state[col] != row:
                successor = list(state)
                successor[col] = row
                successors.append(successor)
    return successors

def heuristicFunction(state):
    """
        Calculates the number of conflicts (queens attacking each other) in
the given state.

        Parameters:
            - state (list): A list representing the current state of the
board.

        Returns:
            - attacks (int): The number of conflicts in the given state.
        """
    n = len(state)
    attacks = 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):
                attacks += 1
    return attacks

def hill_climbing(initial_state):
    """
        Performs hill climbing search to solve the n-queens problem.

        Parameters:
            - initial_state (list): A list representing the initial state
of the board.

        Returns:
            - success (bool): True if the goal state was reached, False
otherwise.
            - current_state (list): A list representing the final state of
the board.
            - moves (int): The number of moves made during the search.
        """
    current_state = initial_state
    moves = 0
    while True:
        successor_states = generateSuccessorStates(current_state)
        if len(successor_states) == 0:
            break
        best_successor = min(successor_states, key=heuristicFunction)
```

```
        if heuristicFunction(best_successor) >=
heuristicFunction(current_state):
            break
        current_state = best_successor
        moves += 1
    if heuristicFunction(current_state) == 0:
        success = True
    else:
        success = False
    return success, current_state, moves

def simulated_annealing(initial_state):
    """
       Performs simulated annealing search to solve the n-queens problem.

       Parameters:
           - initial_state (list): A list representing the initial state of
the board.

       Returns:
           - success (bool): True if the goal state was reached, False
otherwise.
           - current_state (list): A list representing the final state of
the board.
           - moves (int): The number of moves made during the search.
       """
    current_state = initial_state
    temperature = 1.0
    moves = 0
    while temperature > 0.1:
        successor_states = generateSuccessorStates(current_state)
        if len(successor_states) == 0:
            break
        next_state = random.choice(successor_states)
        delta = heuristicFunction(next_state) -
heuristicFunction(current_state)
        if delta < 0 or random.random() < math.exp(-delta / temperature):
            current_state = next_state
        temperature *= 0.99
        moves += 1
    if heuristicFunction(current_state) == 0:
        success = True
    else:
        success = False
    return success, current_state, moves

if __name__ == '__main__':
    # Ask the user for the number of queens and generate a random start
state
    numberOfQueens = int(input("Please enter the number of queens to add on
board (5-10): "))
    initial_state = generateRandomStartState(numberOfQueens)

    # Print the start state of the board
    print("\nStart State: ")
    for i in range(numberOfQueens):
        row = ['-']*numberOfQueens
        row[initial_state[i]] = 'Q'
        print(' '.join(row))

    # Use Hill Climbing algorithm to find a solution and print the result
```

```
    print("\n\nHill Climbing: ")
    success, final_state, moves1 = hill_climbing(initial_state)
    if success:
        print("\nSuccess! (goal state reached)")
    else:
        print("\nFailure! (goal state not reached)")
    print("\nNumber of moves made: ", moves1)

    print("\nFinal State: ")
    for i in range(numberOfQueens):
        row = ['-']*numberOfQueens
        row[final_state[i]] = 'Q'
        print(' '.join(row))

    # Use Simulated Annealing algorithm to find a solution and print the result
    print("\n\nSimulated Annealing: ")
    success, final_state, moves2 = simulated_annealing(initial_state)
    if success:
        print("\nSuccess! (goal state reached)")
    else:
        print("\nFailure! (goal state not reached)")
    print("\nNumber of moves made: ", moves2)

    print("\nFinal State: ")
    for i in range(numberOfQueens):
        row = ['-']*numberOfQueens
        row[final_state[i]] = 'Q'
        print(' '.join(row))

    # Compare the results of Hill Climbing and Simulated Annealing
    if moves1 < moves2:
        print("\nHill Climbing is better. It achieves the best minima.")
    elif moves2 < moves1:
        print("\nSimulated Annealing is better. It achieves the best minima.")
    else:
        print("\nBoth algorithms are equally good.")
```

Output:

N_Queen_Problem

```
C:\Users\hp\Project1\Scripts\python.exe C:\Users\hp\PycharmProjects\firstproject\
Please enter the number of queens to add on board (5-10): 9

Start State:
- - - - - - - Q -
- - - - - - - Q -
- - - - Q - - - -
- - - - - - Q - -
- - - - - - Q - -
- Q - - - - - - -
- - - - - - Q - -
- - Q - - - - - -
- - - - - - - - Q
```

```
Hill Climbing:

Success! (goal state reached)

Number of moves made:  4

Final State:
- - - - Q - - - -
- - - - - - - Q -
- - - Q - - - - -
Q - - - - - - - -
- - - - - - Q - -
- Q - - - - - - -
- - - - - Q - - -
- - Q - - - - - -
- - - - - - - - Q
```

```
Simulated Annealing:

Failure! (goal state not reached)

Number of moves made:  230

Final State:
- - - - - Q - - -
- Q - - - - - - -
- Q - - - - - - -
- - - - - - - Q -
- - Q - - - - - -
- - - - - - Q - -
- - - - - - - - Q
Q - - - - - - - -
- - - - Q - - - -

Hill Climbing is better. It achieves the best minima.

Process finished with exit code 0
```

Problem 2 [Adversarial Search]

a) Minimax algorithm (without pruning)

Code:

```
import random

class TicTacToe:
    """
    The TicTacToe class has a 3x3 board which is initialized to all empty
cells at the start of the game.
    The game starts with a random player (X or O) and each player takes
turns until a win or a draw occurs.
    """
```

```python
    def __init__(self):
        self.board = [['-', '-', '-'], ['-', '-', '-'], ['-', '-', '-']]
        self.player = None
        self.moveCount = 0
        self.maxSuccessors = 0
        self.minSuccessors = 0

    def playGame(self):

        # Randomly decide who starts
        self.player = 'X' if random.randint(0, 1) == 0 else 'O'
        if(self.player == 'X'):
            print("\nMax won the toss and will make first move.")
        else:
            print("\nMin won the toss and will make first move.")

        print(f"{self.player} starts the game!")

        # Continue playing until the game is over
        while True:
            self.showBoard()

            # Check for terminal state
            gameWinner = self.checkWinner()
            if gameWinner is not None:
                self.showBoard()
                print(f"Player {gameWinner} wins the game!")
                break
            elif self.moveCount == 9:
                self.showBoard()
                print("The game is a draw!")
                break

            # Decide which player's turn it is
            if self.player == 'X':
                self.maxTurn()
            else:
                self.minTurn()

    def maxTurn(self):
        """
        The maxTurn method is called when it is the maximizer's turn, i.e.,
the player with the 'X' symbol.
        This method finds the best move using the minimax algorithm and
makes that move.
        """
        print("Max's turn (X):")

        # Find best move using minimax algorithm
        highScore = float('-inf')
        goodMove = None
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == '-':
                    self.board[i][j] = 'X'
                    scoreBoard = self.minimax(0, False)
                    self.board[i][j] = '-'
                    if scoreBoard > highScore:
                        highScore = scoreBoard
                        goodMove = (i, j)
```

```python
        # Make the best move
        self.board[goodMove[0]][goodMove[1]] = 'X'
        self.moveCount += 1
        self.maxSuccessors += 1
        self.player = 'O'

    def minTurn(self):
        """
        The minTurn method is called when it is the minimizer's turn, i.e.,
the player with the 'O' symbol.
        This method finds the best move using the minimax algorithm and
makes that move.
        """
        print("Min's turn (O):")

        # Find best move using minimax algorithm
        highScore = float('inf')
        goodMove = None
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == '-':
                    self.board[i][j] = 'O'
                    scoreBoard = self.minimax(0, True)
                    self.board[i][j] = '-'
                    if scoreBoard < highScore:
                        highScore = scoreBoard
                        goodMove = (i, j)

        # Make the best move
        self.board[goodMove[0]][goodMove[1]] = 'O'
        self.moveCount += 1
        self.minSuccessors += 1
        self.player = 'X'

    def minimax(self, depth, maximizingPlayer):
        """
        The minimax method is a recursive function that implements the
minimax algorithm.
        It takes in the current depth and a boolean flag indicating whether
it is the maximizing or minimizing player's turn.
        The base case of the recursion is when the game is in a terminal
state (win or draw),
        in which case it returns a score of 1, -1, or 0 depending on
whether 'X', 'O', or neither won the game.
        The recursive case involves iterating over all possible moves and
recursively calling minimax with the opposite player's turn.
        The algorithm keeps track of the highest or lowest score depending
on whether it is the maximizing or minimizing player's turn, respectively.
        :param depth:
        :param maximizingPlayer:
        :return:
        """

        # Check for terminal state
        gameWinner = self.checkWinner()
        if gameWinner == 'X':
            return 1
        elif gameWinner == 'O':
            return -1
        elif self.moveCount == 9:
            return 0
```

```
        # Recursive case
        if maximizingPlayer:
            highScore = float('-inf')
            for i in range(3):
                for j in range(3):
                    if self.board[i][j] == '-':
                        self.board[i][j] = 'X'
                        scoreBoard = self.minimax(depth + 1, False)
                        self.board[i][j] = '-'
                        highScore = max(scoreBoard, highScore)
            return highScore
        else:
            highScore = float('inf')
            for i in range(3):
                for j in range(3):
                    if self.board[i][j] == '-':
                        self.board[i][j] = 'O'
                        scoreBoard = self.minimax(depth + 1, True)
                        self.board[i][j] = '-'
                        highScore = min(scoreBoard, highScore)
            return highScore

    def checkWinner(self):
        """
        The checkWinner method checks if there is a winner in the current
game state.
        It checks rows, columns, and diagonals for three consecutive
symbols.
        """
        # Check rows for winner
        for row in self.board:
            if row == ['X', 'X', 'X']:
                return 'X'
            elif row == ['O', 'O', 'O']:
                return 'O'

        # Check columns for winner
        for j in range(3):
            col = [self.board[i][j] for i in range(3)]
            if col == ['X', 'X', 'X']:
                return 'X'
            elif col == ['O', 'O', 'O']:
                return 'O'

        # Check diagonals for winner
        diagonalOne = [self.board[i][i] for i in range(3)]
        diagonalTwo = [self.board[i][2 - i] for i in range(3)]
        if diagonalOne == ['X', 'X', 'X'] or diagonalTwo == ['X', 'X', 'X']:
            return 'X'
        elif diagonalOne == ['O', 'O', 'O'] or diagonalTwo == ['O', 'O',
'O']:
            return 'O'

        return None

    def showBoard(self):
        """
        The showBoard method displays the current state of the board in a
human-readable format.
        """
```

```python
        for i in range(3):
            print("+---+---+---+")
            row = "| "
            for j in range(3):
                row += self.board[i][j] + " | "
            print(row)
        print("+---+---+---+")
        print('\n')

if __name__ == '__main__':
    print("Welcome to Tic Tac Toe!")
    print("Let's start the game!")
    t = TicTacToe()
    t.playGame()
    print("\nTotal moves taken: ",t.moveCount)
    print("\nTotal successors generated by Min: ",t.minSuccessors)
    print("\nTotal successors generated by Max: ",t.maxSuccessors)
```

Output:

```
C:\Users\hp\Project1\Scripts\python.exe C:\Users\hp\PycharmProjects\firstproject\A
Welcome to Tic Tac Toe!
Let's start the game!

Max won the toss and will make first move.
X starts the game!
+---+---+---+
| - | - | - |
+---+---+---+
| - | - | - |
+---+---+---+
| - | - | - |
+---+---+---+


Max's turn (X):
+---+---+---+
| X | - | - |
+---+---+---+
| - | - | - |
+---+---+---+
| - | - | - |
+---+---+---+
```

```
Min's turn (O):
+---+---+---+
| X | O | - |
+---+---+---+
| - | - | - |
+---+---+---+
| - | - | - |
+---+---+---+


Max's turn (X):
+---+---+---+
| X | O | X |
+---+---+---+
| - | - | - |
+---+---+---+
| - | - | - |
+---+---+---+
```

```
Min's turn (O):
+---+---+---+
| X | O | X |
+---+---+---+
| O | - | - |
+---+---+---+
| - | - | - |
+---+---+---+


Max's turn (X):
+---+---+---+
| X | O | X |
+---+---+---+
| O | X | - |
+---+---+---+
| - | - | - |
+---+---+---+
```

```
Min's turn (O):
+---+---+---+
| X | O | X |
+---+---+---+
| O | X | O |
+---+---+---+
| - | - | - |
+---+---+---+


Max's turn (X):
+---+---+---+
| X | O | X |
+---+---+---+
| O | X | O |
+---+---+---+
| X | - | - |
+---+---+---+
```

```
+---+---+---+
| X | O | X |
+---+---+---+
| O | X | O |
+---+---+---+
| X | - | - |
+---+---+---+


Player X wins the game!

Total moves taken:  7

Total successors generated by Min:  3

Total successors generated by Max:  4

Process finished with exit code 0
```

b) Minimax algorithm (with alpha/beta pruning)

Code:

```
import random

class TicTacToe:
    """
    The TicTacToe class has a 3x3 board which is initialized to all empty
cells at the start of the game.
    The game starts with a random player (X or O) and each player takes
turns until a win or a draw occurs.
    """
    def __init__(self):
        self.board = [['-', '-', '-'], ['-', '-', '-'], ['-', '-', '-']]
        self.player = None
        self.moveCount = 0
        self.maxSuccessors = 0
        self.minSuccessors = 0
```

```
    def playGame(self):

        # Randomly decide who starts
        self.player = 'X' if random.randint(0, 1) == 0 else 'O'
        if(self.player == 'X'):
            print("\nMax won the toss and will make first move.")
        else:
            print("\nMin won the toss and will make first move.")

        print(f"{self.player} starts the game!")

        # Continue playing until the game is over
        while True:
            self.showBoard()

            # Check for terminal state
            gameWinner = self.checkWinner()
            if gameWinner is not None:
                self.showBoard()
                print(f"Player {gameWinner} wins the game!")
                break
            elif self.moveCount == 9:
                self.showBoard()
                print("The game is a draw!")
                break

            # Decide which player's turn it is
            if self.player == 'X':
                self.maxTurn()
            else:
                self.minTurn()

    def maxTurn(self):
        """
        The maxTurn method is called when it is the maximizer's turn, i.e.,
the player with the 'X' symbol.
        This method finds the best move using the minimax algorithm and
makes that move.
        """
        print("Max's turn (X):")

        # Find best move using minimax algorithm
        highScore = float('-inf')
        goodMove = None
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == '-':
                    self.board[i][j] = 'X'
                    scoreBoard = self.minimax(0, float('-inf'), float('inf'),
False)

                    self.board[i][j] = '-'
                    if scoreBoard > highScore:
                        highScore = scoreBoard
                        goodMove = (i, j)

        # Make the best move
        self.board[goodMove[0]][goodMove[1]] = 'X'
        self.moveCount += 1
        self.maxSuccessors += 1
        self.player = 'O'
```

```python
    def minTurn(self):
        """
        The minTurn method is called when it is the minimizer's turn, i.e.,
the player with the 'O' symbol.
        This method finds the best move using the minimax algorithm and
makes that move.
        """
        print("Min's turn (O):")

        # Find best move using minimax algorithm
        highScore = float('inf')
        goodMove = None
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == '-':
                    self.board[i][j] = 'O'
                    scoreBoard = self.minimax(0, float('-inf'), float('inf'),
True)
                    self.board[i][j] = '-'
                    if scoreBoard < highScore:
                        highScore = scoreBoard
                        goodMove = (i, j)

        # Make the best move
        self.board[goodMove[0]][goodMove[1]] = 'O'
        self.moveCount += 1
        self.minSuccessors += 1
        self.player = 'X'

    def minimax(self, depth, alpha, beta, maximizingPlayer):
        """
        The minimax method is a recursive function that implements the
minimax algorithm with alpha-beta pruning.
        It takes in the current depth, alpha, beta and a boolean flag
indicating whether it is the maximizing or minimizing player's turn.
        The base case of the recursion is when the game is in a terminal
state (win or draw),
        in which case it returns a score of 1, -1, or 0 depending on
whether 'X', 'O', or neither won the game.
        The recursive case involves iterating over all possible moves and
recursively calling minimax with the opposite player's turn.
        The algorithm keeps track of the highest or lowest score depending
on whether it is the maximizing or minimizing player's turn, respectively.
        :param depth:
        :param alpha:
        :param beta:
        :param maximizingPlayer:
        :return:
        """

        # Check for terminal state
        gameWinner = self.checkWinner()
        if gameWinner == 'X':
            return 1
        elif gameWinner == 'O':
            return -1
        elif self.moveCount == 9:
            return 0

        # Recursive case
```

```python
        if maximizingPlayer:
            highScore = float('-inf')
            for i in range(3):
                for j in range(3):
                    if self.board[i][j] == '-':
                        self.board[i][j] = 'X'
                        scoreBoard = self.minimax(depth + 1, alpha, beta,
False)

                        self.board[i][j] = '-'
                        highScore = max(scoreBoard, highScore)
                        alpha = max(alpha, highScore)
                        if beta <= alpha:
                            break
            return highScore
        else:
            lowScore = float('inf')
            for i in range(3):
                for j in range(3):
                    if self.board[i][j] == '-':
                        self.board[i][j] = 'O'
                        scoreBoard = self.minimax(depth + 1, alpha, beta,
True)

                        self.board[i][j] = '-'
                        lowScore = min(scoreBoard, lowScore)
                        beta = min(beta, lowScore)
                        if beta <= alpha:
                            break
            return lowScore

    def checkWinner(self):
        """
        The checkWinner method checks if there is a winner in the current
game state.
        It checks rows, columns, and diagonals for three consecutive
symbols.
        """
        # Check rows for winner
        for row in self.board:
            if row == ['X', 'X', 'X']:
                return 'X'
            elif row == ['O', 'O', 'O']:
                return 'O'

        # Check columns for winner
        for j in range(3):
            col = [self.board[i][j] for i in range(3)]
            if col == ['X', 'X', 'X']:
                return 'X'
            elif col == ['O', 'O', 'O']:
                return 'O'

        # Check diagonals for winner
        diagonalOne = [self.board[i][i] for i in range(3)]
        diagonalTwo = [self.board[i][2 - i] for i in range(3)]
        if diagonalOne == ['X', 'X', 'X'] or diagonalTwo == ['X', 'X', 'X']:
            return 'X'
        elif diagonalOne == ['O', 'O', 'O'] or diagonalTwo == ['O', 'O',
'O']:
            return 'O'

        return None
```

```
    def showBoard(self):
        """
        The showBoard method displays the current state of the board in a
human-readable format.
        """
        for i in range(3):
            print("+---+---+---+")
            row = "| "
            for j in range(3):
                row += self.board[i][j] + " | "
            print(row)
        print("+---+---+---+")
        print('\n')

if __name__ == '__main__':
    print("Welcome to Tic Tac Toe!")
    print("Let's start the game!")
    print("Using alpha beta pruning!")
    t = TicTacToe()
    t.playGame()
    print("\nTotal moves taken: ",t.moveCount)
    print("\nTotal successors generated by Min: ",t.minSuccessors)
    print("\nTotal successors generated by Max: ",t.maxSuccessors)
```

Output:

```
Welcome to Tic Tac Toe!
Let's start the game!
Using alpha beta pruning!

Max won the toss and will make first move.
X starts the game!
+---+---+---+
| - | - | - |
+---+---+---+
| - | - | - |
+---+---+---+
| - | - | - |
+---+---+---+


Max's turn (X):
+---+---+---+
| X | - | - |
+---+---+---+
| - | - | - |
+---+---+---+
| - | - | - |
+---+---+---+
```

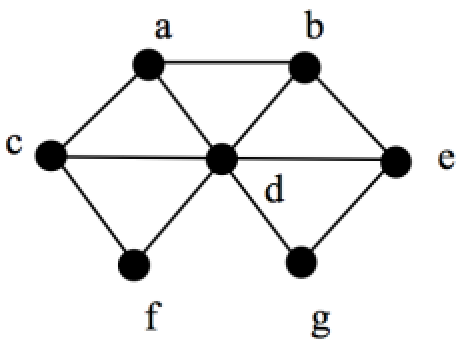
```
Min's turn (O):
+---+---+---+
| X | O | - |
+---+---+---+
| - | - | - |
+---+---+---+
| - | - | - |
+---+---+---+


Max's turn (X):
+---+---+---+
| X | O | X |
+---+---+---+
| - | - | - |
+---+---+---+
| - | - | - |
+---+---+---+
```

```
Min's turn (O):
+---+---+---+
| X | O | X |
+---+---+---+
| O | - | - |
+---+---+---+
| - | - | - |
+---+---+---+


Max's turn (X):
+---+---+---+
| X | O | X |
+---+---+---+
| O | X | - |
+---+---+---+
| - | - | - |
+---+---+---+
```

```
Min's turn (O):
+---+---+---+
| X | O | X |
+---+---+---+
| O | X | O |
+---+---+---+
| - | - | - |
+---+---+---+


Max's turn (X):
+---+---+---+
| X | O | X |
+---+---+---+
| O | X | O |
+---+---+---+
| X | - | - |
+---+---+---+
```

```
+---+---+---+
| X | O | X |
+---+---+---+
| O | X | O |
+---+---+---+
| X | - | - |
+---+---+---+


Player X wins the game!

Total moves taken:  7

Total successors generated by Min:  3

Total successors generated by Max:  4

Process finished with exit code 0
```

Problem 3 [Constraint Satisfaction Problem]



a) AC-3

Code:

```python
from queue import Queue
from collections import deque

# Define the graph as an adjacency list
graph = {
    'a': ['b', 'c','d'],
    'b': ['a', 'd', 'e'],
    'c': ['a', 'f', 'd'],
    'd': ['a','b', 'c', 'e', 'f','g'],
    'e': ['b', 'd', 'g'],
    'f': ['d','c'],
    'g': ['e', 'd']
}

# Define the domain of colors
colors = ['R', 'G', 'B']

# Variable and domain definition
variables = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
domain = {variable: ['R', 'G', 'B'] for variable in variables}


# Define the AC-3 algorithm
def AC_3(queueArr, domainValue, constraints):
    while not queueArr.empty():
        (i, j) = queueArr.get()
        if revise(domainValue, i, j):
            if len(domainValue[i]) == 0:
                return False
            for k in graph[i]:
                if k != j:
                    queueArr.put((k, i))
    return True



# Constraint function
def isConsistent(variable, value, assignment):
    for neighbor in graph[variable]:
        if neighbor in assignment and assignment[neighbor] == value:
            return False
    return True


# AC-3 algorithm
def ac3(queue, domains, graph):
    num_assignments = 0
    while queue:
        (xi, xj) = queue.popleft()
        if revise(domains, xi, xj):
            num_assignments += 1
            if len(domains[xi]) == 0:
                return False, num_assignments
            for xk in graph[xi]:
                if xk != xj:
                    queue.append((xk, xi))
    return True, num_assignments


def revise(domains, xi, xj):
    revised = False
```

```
    for vi in domains[xi]:
        has_support = False
        for vj in domains[xj]:
            if isConsistent(xi, vi, {xj: vj}):
                has_support = True
                break
        if not has_support:
            domains[xi].remove(vi)
            revised = True
    return revised


# Backtracking algorithm
def backTrack(assignment):
    if len(assignment) == len(variables):
        return assignment

    variable = None
    # Chronological order heuristic
    for var in variables:
        if var not in assignment:
            variable = var
            break

    for domainValue in domain[variable]:
        if isConsistent(variable, domainValue, assignment):
            assignment[variable] = domainValue
            result, num_assignments = ac3(deque([(x, variable) for x in
graph[variable] if x not in assignment]), domain.copy(),
                                          graph)
            if result:
                result = backTrack(assignment)
                if result:
                    return result
            num_assignments += 1
            del assignment[variable]
            for x in graph[variable]:
                if x not in assignment:
                    domain[x].append(domainValue)
            domain[variable] = ['R', 'G', 'B']
    return False

# Define the main function that solves the problem
def graphColoring(graph, colors, domainValue, constraints):
    queueArr = Queue()
    for i in graph:
        for j in graph[i]:
            queueArr.put((i, j))
    if AC_3(queueArr, domainValue, constraints):
        print("\nAC-3 succeeded.")
        assignments = 0
        for variable in domainValue:
            print("Variable:", variable, "\tDomain:", domainValue[variable])
            assignments += 1
    else:
        print("AC-3 failed.")

# Call the main function to solve the problem
graphColoring(graph, colors, domain, lambda x, y: x != y)
```

```python
# Solving the problem
assignedValue = {}
result = backTrack(assignedValue)

# Outputting results
if result:
    print("\nYes! A solution exists: ")
    print("\nValues of all variables after assignment: ")
    for var in variables:
        print("Variable",var, "=", result[var])
    print("\nTotal number of assignments made: ", len(result))

else:
    print("No solution exists!")
```

Output:

```
C:\Users\hp\Project1\Scripts\python.exe C:\Users\hp\PycharmProjects\firstproject\GraphCo

AC-3 succeeded.
Variable: a     Domain: ['R', 'G', 'B']
Variable: b     Domain: ['R', 'G', 'B']
Variable: c     Domain: ['R', 'G', 'B']
Variable: d     Domain: ['R', 'G', 'B']
Variable: e     Domain: ['R', 'G', 'B']
Variable: f     Domain: ['R', 'G', 'B']
Variable: g     Domain: ['R', 'G', 'B']

Yes! A solution exists:

Values of all variables after assignment:
Variable a = R
Variable b = G
Variable c = G
Variable d = B
Variable e = R
Variable f = R
Variable g = G

Total number of assignments made:  7

Process finished with exit code 0
```

b)DFS with backtracking, MRV, DH and LCV

Code:

```
from collections import deque

# Define the graph as an adjacency list
graph = {
    'a': ['b', 'c', 'd'],
    'b': ['a', 'd', 'e'],
    'c': ['a', 'f', 'd'],
    'd': ['a', 'b', 'c', 'e', 'f', 'g'],
    'e': ['b', 'd', 'g'],
    'f': ['d', 'c'],
    'g': ['e', 'd']
}

# Define the domain of colors
colors = ['R', 'G', 'B']

# Variable and domain definition
variables = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
domain = {variable: ['R', 'G', 'B'] for variable in variables}

# Constraint function
def isConsistent(variable, value, assignment):
    for neighbor in graph[variable]:
        if neighbor in assignment and assignment[neighbor] == value:
            return False
    return True

# Backtracking algorithm with DFS and heuristics
def backTrack(assignment):
    if len(assignment) == len(variables):
        return assignment

    # Minimum Remaining Value (MRV) heuristic
    unassigned_vars = [var for var in variables if var not in assignment]
    mrv_vars = sorted(unassigned_vars, key=lambda var: len(domain[var]))

    variable = None
    if len(mrv_vars) > 0:
        variable = mrv_vars[0]

    # Degree Heuristic (DH)
    if variable is not None:
        dh_vars = sorted([var for var in graph[variable] if var not in
assignment], key=lambda var: len(graph[var]))
        if len(dh_vars) > 0:
            variable = dh_vars[0]

    # Least Constraining Value (LCV) heuristic
    lcv_values = []
    for domainValue in domain[variable]:
        num_conflicts = 0
        for neighbor in graph[variable]:
            if neighbor in assignment and assignment[neighbor] !=
domainValue:
                num_conflicts += 1
        lcv_values.append((domainValue, num_conflicts))
    lcv_values.sort(key=lambda x: x[1])
```

```python
    for domainValue, _ in lcv_values:
        if isConsistent(variable, domainValue, assignment):
            assignment[variable] = domainValue
            result = backTrack(assignment)
            if result:
                return result
            del assignment[variable]

    return False

# Define the main function that solves the problem
def graphColoring(graph, colors, domain, constraints):
    # Call backtracking algorithm with heuristics
    assignedValue = {}
    result = backTrack(assignedValue)

    # Outputting results
    if result:
        print("\nUsing DFS with backtracking, MRV, LCV and DH: ")
        print("\nYes! A solution exists: ")
        print("\nValues of all variables after assignment: ")
        for var in variables:
            print("Variable", var, "=", result[var])
        print("\nTotal number of assignments made: ", len(result))
    else:
        print("No solution exists!")

# Call the main function to solve the problem
graphColoring(graph, colors, domain, lambda x, y: x != y)
```

Output:

```
C:\Users\hp\Project1\Scripts\python.exe C:\Users\hp\Pycha

Using DFS with backtracking, MRV, LCV and DH:

Yes! A solution exists:

Values of all variables after assignment:
Variable a = B
Variable b = R
Variable c = R
Variable d = G
Variable e = B
Variable f = B
Variable g = R

Total number of assignments made:  7

Process finished with exit code 0
```

The End
