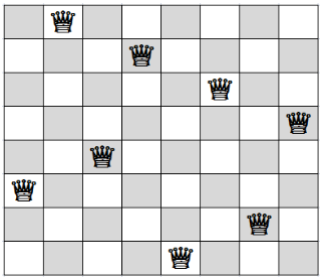
UNC CHARLOTTE
College of Computing and Informatics

PROJECT DOCUMENTATION REPORT

ON

Solving N-Queens Problem by Hill-Climbing and its variants



PROGRAMMING PROJECT 2
ITCS 6150 - Intelligent Systems

DEPARTMENT OF COMPUTER SCIENCE

SUBMITTED TO
Dewan T. Ahmed, Ph.D.

SUBMITTED BY:

Urma Haldar
801074045

Sahil Sood
801082267

Table of Contents

PROBLEM FORMULATION

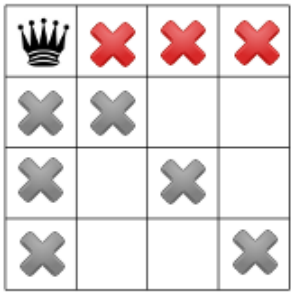
1.1 INTRODUCTION

What is N – Queen Problem?

The *N*-queens problem originally introduced in 1850 by Carl Gauss.

The goal of this problem is to place N queens on a $N \times N$ chessboard, so that no queens can take each other. Queens can move horizontally, vertically, and diagonally, this means that there can be only one queen per row and one per column, and that no two queens can find themselves on the same diagonal.

It is a computationally expensive problem - NP-complete, what makes it very famous problem in computer science.



Hill Climbing Search:

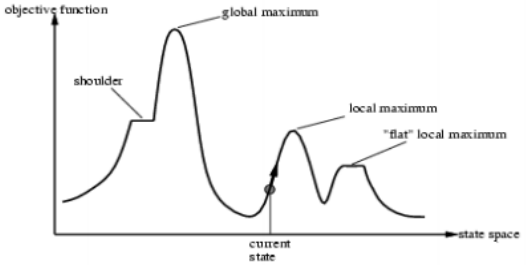
Hill Climbing is heuristic search used for mathematical optimization problems in the field of Artificial Intelligence.

Given a large set of inputs and a good heuristic function, it tries to find a sufficiently good solution to the problem. This solution may not be the global optimal maximum.

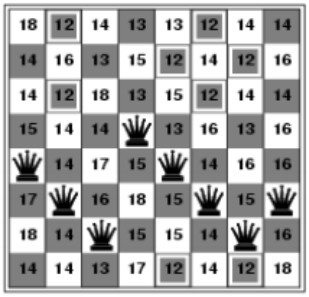
- In the above definition, mathematical optimization problems implies that hill climbing solves the problems where we need to maximize or minimize a given real function by choosing values from the given inputs.
- 'Heuristic search' means that this search algorithm may not find the optimal solution to the problem. However, it will give a good solution in reasonable time.
- A heuristic function is a function that will rank all the possible alternatives at any branching step in search algorithm based on the available information. It helps the algorithm to select the best route out of possible routes.

Hill-climbing search

- Problem: depending on initial state, can get stuck in local maxima



Hill-climbing search: 8-queens problem



- *h* = number of pairs of queens that are attacking each other, either directly or indirectly
- *h* = 17 for the above state

Steepest Ascent Hill Climbing

This differs from the basic Hill climbing algorithm by choosing the best successor rather than the first successor that is better. This indicates that it has elements of the breadth first algorithm.

Both the basic and this method of hill climbing may fail to find a solution by reaching a state from which no subsequent improvement can be made and this state is not the solution.

Hill Climbing with Sideways move

Hill-climbing : how to avoid shoulders ?
Starting from a random initial 8-queens state, hill-climbing gets stuck in a local maximum 86% of the time (with an average of 3 steps) and finds the global maximum only 14% (with an average of 4 steps). Sideways moves To avoid staying stuck in a shoulder plateau, we allow hill-climbing to move to neighbour states that have the same value as the current one. However, infinite loops may occur. We avoid infinite loops by limiting the number of sideways moves.

By allowing up to 100 sideways moves in the 8-queens problem, We increase the percentage of problem instances solved by hill climbing from 14% to 94%. However, each successful instance is solved in 21 steps (on average) and failures (local maxima) are reached after 64 steps (on average).

Random Restart Hill Climbing

Random-restart hill-climbing Repeats the hill-climbing from a randomly generated initial state, until a solution is found. Guaranteed to eventually find a solution if the state space is finite. The expected number of restarts is 1 p if hill-climbing succeeds with probability p at each iteration.

Example of random-restart hill-climbing in the 8-queens problem Without sideways moves : p = 0.14, we need about 7 iterations on average to find a solution. The average number of steps is 22. With sideways moves : p = 0.94, we need about 1.06 iterations on average to find a solution. The average number of steps is 22.

1.2 ALGORITHM PSEUDOCODE

```
function HILL-CLIMBING(problem) returns a solution state
    inputs: problem, a problem
    static: current, a node
       next, a node
    current <— MAKE-NODE(INITIAL-STATE[problem])
    loop do
       next— a highest-valued successor of current
       if VALUE[next] < VALUE[current] then return current
       current *—next
    end
```

1.3 EXPECTED OUTPUT

With Steepest Ascent, we are expecting 14% success and 96% failure rates, with 3 steps to success and 4 steps to failure.

With Sideways moves, we are expecting 96% success rate and 4% failure rate with 22 steps to success and 64 steps to failure.

With random restart, we get 100% success

PROGRAM STRUCTURE

2.1 GLOBAL/LOCAL VARIABLES

We use various global/local variables in our program:
The local variables within function are:

- hits – calculates the number of row collisions as we as diagonal collisions for a particular cell
- i, j – for the range
- row – variable that stores the row index
- col – variable that stores the column index
- checkBetter – flag variable that stores true or false when a better solution is found
- best – duplicate array to handle different operations
- mins -- variable that stores the collisions of current board so that if finds better than this it will quit
- store – this is an array list that stores the columns from which a random column will be selected
- randomCount – counter variable that stores the number of random restarts
- movesTotal – variable that stores the total number of moves
- movesSolution – variable that stores the total number of moves in the solution set
- maxSteps – variable that stores the maximum steps of the iteration
- Iterations – variable that stores the number of iterations for sideways move hill climbing
- min_compare – variable to compare the position in column with the minimum conflicts
- randomValue -- this stores the current queue position in the randomly selected column
- currentValue – stores the value of current column
- Index – stores the index value of a particular column on the board

- The global variables within function are:

- choice – variable that stores the input value from the user for a particular operation that it chooses
- a – array that stores the initial configuration of the board
- b – duplicate array that stores the same value as variable a
- queen – object variable of the class NQueenProblem

2.2 FUNCTIONS/PROCEDURES

Our code implements n queens problem using Hill Climbing local search method and its variants – Steepest-ascent, Hill Climbing with Sideways move and Random restart Hill Climbing using Python.

We use the following functions within the code:

- rowHits() - This method calculates all the row conflicts for a queen placed in a particular cell.
- diagonalHits() - This method calculates all the diagonal conflicts for a particular position of the queen
- totalHits() - This method returns total number of collisions for a particular queen position
- optimalSol() - This method calculates the conflicts for the current state of the board and quits whenever finds a better state.
- randomGen() – This function generates a random state of the board
- checkSol() – This method verifies whether the current state of the board is the solution(I.e. with zero conflicts)
- minHits() - This method finds the solution for the n-queens problem with Min-Conflicts algorithm with random restart
- minHitswithoutRestart() - This method finds the solution for the n-queens problem with Min-Conflicts algorithm without random restart
- collisionsMinHits() - This method returns the collisions of a queen in a particular column of the board
- fillList() - This function fills the Array List with numbers 0 to n-1

2.3 LOGIC

We are running the code in python 3 in Anaconda package Spyder idle. We first generate a random board with n queens in it. The functions row collisions and column collisions count the number of attacks and keep a track of it. The movements are made as per the hill climbing technique used (Steepest Ascent or Sideways moves) and collisions are compared and accordingly the next step is taken. With random restart technique, on failure, again a random board gets generated and it starts looking for a solution, hence a solution is guaranteed. We print the steps, success failure rates at the end. For Sideways moves technique, we put a **limitation over sideways moves to stop it from entering an infinite loop**. The more queens, the more steps. The more iterations, the more steps.

Commenting the print of the sequences as there are minimum 20 moves on success and 64 moves on failure in Sideway moves, might let the system/kernel to crash.

Calculation of average steps for success – Total number of success moves/ Total number of success
Calculation of average steps for failure – Total number of failure moves/ Total number of failure
Success/Failure rate = (Success/Failure)/Total * 100

PROBLEM CODE

3.1 Code for Steepest Ascent Hill Climbing

With Restart:

```
import random
import copy

class NQueenProblem(object):
    # This method calculates all the row conflicts for a queen placed in a
particular cell.
    @classmethod
    def rowHits(self, a, n):
        hits = 0
        i = 0
        while i < n:
            j = 0
            while j < n:
                if i != j:
                    if a[i] == a[j]:
                        hits += 1
                j += 1
            i += 1
        return hits

    # This method calculates all the diagonal conflicts for a particular
position of the queen
    @classmethod
    def diagonalHits(self, a, n):
        hits = 0
        d = 0
        i = 0
        while i < n:
            j = 0
            while j < n:
                if i != j:
```

```
                    d = abs(i - j)
                    if (a[i] == a[j] + d) or (a[i] == a[j] - d):
                        hits += 1
                j += 1
            i += 1
        return hits

    # This method returns total number of hits for a particular queen
position
    @classmethod
    def totalHits(self, a, n):
        hits = 0
        hits = self.rowHits(a, n) + self.diagonalHits(a, n)
        return hits

    # This method calculates the conflicts for the current state of the board
and quits whenever finds a better state.
    @classmethod
    def optimalSol(self, a, n):
        hits = 0
        row = -1
        col = -1
        checkBetter = False
        best = []
        # Sets min variable to the hits of current board so that if finds
better than this it will quit.
        mins = self.totalHits(a, n)
        best = copy.deepcopy(a)
        # Create a duplicate array for handling different operations
        i = 0
        while i < n:
            # This iteration is for each column
            if checkBetter:
                # If it finds and better state than the current, it will quit
                break
            m = best[i]
            j = 0
            while j < n:
```

```
                # This iteration is for each row in the selected column
                if j != m:
                    # This condition ensures that, current queen position is
not taken into consideration.
                    best[i] = j
                    # Assigning the queen to each position and then
calculating the hits
                    hits = self.totalHits(best, n)
                    if mins > hits:
                        # If a better state is found, that particular column
and row values are stored
                        col = i
                        row = j
                        mins = hits
                        checkBetter = True
                        break
                best[i] = m
                # Restoring the array to the current board position
                j += 1

            i += 1
        if col == -1 or row == -1:
            # If there is no better state found
            print("Stuck at Local Maxima with " ,hits ,"hits. now using
random restart...")
            return False
        a[col] = row
        return True

    @classmethod
    def printBoard(self,best,n):
        i = 0
        while i < n:
            j = 0
            while j < n:
                if j == best[i]:
                    print(" Q ", end="")
                else:
```

```
                    print(" - ", end="")
                j += 1
            print()
            i += 1

    # Below function generates a random state of the board
    @classmethod
    def randomGen(self, a, n):
        i = 0
        while( i < n):
            a[i] =random.randint(0,n-1) + 0
            i += 1


    # Below function verifies whether the current state of the board is the
solution(I.e with zero conflicts)
    @classmethod
    def checkSol(self, a, n):
        if self.totalHits(a, n) == 0:
            return True
        return False

def main():

    totalRestart = 0
    movesTotal = 0
    movesSolution = 0


        # Randomly generate the board

        # The below code will be executed if the user chooses he options 1 or
3(n-queen with Hill Climbing method)
    n = int(input("Enter the number of Queens on the board: "))

    a = [None] * n
    queens = NQueenProblem()
    queens.randomGen(a, n)
```

```
    while not queens.checkSol(a, n):
                # Executes until a solution is found
        if queens.optimalSol(a, n):
                    # If a better state for a board is found
            movesTotal += 1
            movesSolution += 1
            continue
        else:
                    # If a better state is not found
            movesSolution = 0
            queens.randomGen(a, n)
                    # Board is generated Randomly
            totalRestart += 1
    print("Total number of Restarts: ",totalRestart)
    print("Total number of moves taken: ",movesTotal-1)
    # Gives the total number of moves from starting point
    print("Number of moves in the solution set: ",movesSolution)
    # Gives number of steps in the solution set.
    i = 0
    while i < n:
        j = 0
        while j < n:
            if j == a[i]:
                print(" Q ", end="")
            else:
                print(" - ", end="")
            j += 1
        print()
        i += 1
    if(totalRestart == 0):
        print("Average steps",movesTotal)
    else:
        print("Average steps",movesTotal/totalRestart)

if __name__ == '__main__':
    main()
```

Without Restart:

```
import random
import copy

class NQueenProblem(object):
    # This method calculates all the row conflicts for a queen placed in a
particular cell.
    @classmethod
    def rowHits(self, a, n):
        hits = 0
        i = 0
        while i < n:
            j = 0
            while j < n:
                if i != j:
                    if a[i] == a[j]:
                        hits += 1
                j += 1
            i += 1
        return hits

    # This method calculates all the diagonal conflicts for a particular
position of the queen
    @classmethod
    def diagonalHits(self, a, n):
        hits = 0
        d = 0
        i = 0
        while i < n:
            j = 0
            while j < n:
                if i != j:
```

```
                    d = abs(i - j)
                    if (a[i] == a[j] + d) or (a[i] == a[j] - d):
                        hits += 1
                j += 1
            i += 1
        return hits


    @classmethod
    def printBoard(self,best,n):
        i = 0
        while i < n:
            j = 0
            while j < n:
                if j == best[i]:
                    print(" Q ", end="")
                else:
                    print(" - ", end="")
                j += 1
            print()
            i += 1

    # This method returns total number of hits for a particular queen
position
    @classmethod
    def totalHits(self, a, n):
        hits = 0
        hits = self.rowHits(a, n) + self.diagonalHits(a, n)
        return hits

    # This method calculates the conflicts for the current state of the board
and quits whenever finds a better state.
    #        Note: This function is used for Hill Climbing algorithm
    @classmethod
    def optimalSol(self, a, n):
        global movesTotal
        hits = 0
        row = -1
```

```
        col = -1
        checkBetter = False
        best = []
        # Sets min variable to the hits of current board so that if finds
better than this it will quit.
        mins = self.totalHits(a, n)
        best = copy.deepcopy(a)
        # Create a duplicate array for handling different operations
        i = 0
        while i < n:
            # This iteration is for each column
            if checkBetter:
                # If it finds and better state than the current, it will quit
                break
            m = best[i]
            j = 0
            while j < n:
                # This iteration is for each row in the selected column
                if j != m:
                    # This condition ensures that, current queen position is
not taken into consideration.
                    best[i] = j
                    self.printBoard(best,n)
                    movesTotal +=1
                    print('Queen Position',best)
                    # Assigning the queen to each position and then
calculating the hits
                    hits = self.totalHits(best, n)
                    if mins > hits:
                        # If a better state is found, that particular column
and row values are stored
                        col = i
                        row = j
                        mins = hits
                        checkBetter = True
                        break
                best[i] = m
                # Restoring the array to the current board position
```

```
                j += 1
            i += 1
        if col == -1 or row == -1:
            # If there is no better state found
            print("Stuck at Local Maxima with " ,hits ," hits. Now using
Random restart...")
            return False
        a[col] = row
        return True
        # Returns true to the main function if there is any better state
found

    # Below function generates a random state of the board
    @classmethod
    def randomGen(self, a, n):
        i = 0
        while( i < n):
            a[i] =random.randint(0,n-1) + 0
            i += 1


    # Below function verifies whether the current state of the board is the
solution(I.e with zero conflicts)
    @classmethod
    def checkSol(self, a, n):
        if self.totalHits(a, n) == 0:
            return True
        return False

    # Below method finds the solution for the n-queens problem with Min-
Conflicts algorithm
    @classmethod
    def minConflict(self, b, n, iterations):
        store = []
        self.fillList(store, n)
        randomCount = 0
        movesTotal = 0
        movesSolution = 0
```

```
        row = 0
        maxSteps = iterations
        # The maximum steps that can be allowed to find a solution with this
algorithm
        while not self.checkSol(b, n):
            # Loops until it finds a solution,
            randomSelection = random.randint(0,len(store)-1) + 0
            # Randomly selects a column from the available
            currentValue = b[store[randomSelection]]
            # This stores the current queue position in the randomly selected
column
            randomValue = store[randomSelection]
            mins = self.hitsMinConflict(b, n, randomValue)
            # Sets the minimum variable to the current queue hits
            min_compare = mins
            while(not store):
                store.remove(randomSelection)
            i = 0
            while i < n:
                if currentValue != i:
                    b[randomValue] = i
                    col = self.hitsMinConflict(b, n, randomValue)
                    # Calculates the hits of the queen at particular position
                    if col < mins:
                        mins = col
                        row = i
                i += 1
            if min_compare == mins:
                # When there is no queen with minimum conflicts than the
current position
                if maxSteps != 0:
                    # Checks if the maximum steps is reached
                    if len(store) >= 0:
                        # checks whether there are columns available in the
Array List
                        b[randomValue] = currentValue
                        # restores the queen back to the previous position
                        maxSteps -= 1
```

```
                    else:
                        self.fillList(store, n)
                else:
                    # If the max steps is reached then, the board is
regenerated and initiated the max steps variable
                    randomCount += 1
                    movesSolution = 0
                    self.randomGen(b, n)
                    self.fillList(store, n)
                    maxSteps = iterations
            else:
                # When we find the the position in the column with minimum
conflicts
                movesSolution += 1
                b[randomValue] = row
                min_compare = mins
                store.clear()
                maxSteps -= 1
                self.fillList(store, n)
        print()
        i = 0
        count = 0
        while i < n:
            j = 0
            while j < n:
                if j == b[i]:
                    print(" Q ", end="")
                else:
                    print(" - ", end="")
                j += 1

            print()
            i += 1

        print("Number of Moves in the solution set: ", movesSolution)
```

```
    # Below function returns the hits of a queen in a particular column of
the board
    @classmethod
    def hitsMinConflict(self, b, n, index):
        hits = 0
        t = 0
        i = 0
        while i < n:
            if i != index:
                t = abs(index - i)
                if b[i] == b[index]:
                    hits += 1
                elif b[index] == b[i] + t or b[index] == b[i] - t:
                    hits += 1
            i += 1
        return hits

    # Below function fills the Array List with numbers 0 to n-1
    @classmethod
    def fillList(self, store, n):
        i = 0
        while i < n:
            store.append(i)
            i += 1
        return

def main():
    n = int(input("Enter the number of Queens on the board: "))
    queens = NQueenProblem()
    a = [None] * n
    queens.randomGen(a, n)
    totalRestart = 0
    movesTotal = 0
    movesSolution = 0

    print("How many times do you want to run the code??")
    success=0
    failure=0
```

```
    notimes = int(input("Please enter the value:"))
    while (notimes!=0):
        if queens.optimalSol(a, n):
            movesTotal += 1
            movesSolution += 1
            success+=1
            print('Solution Found')
        else:
            failure+=1
            print('No solution')
        print("Number of Restarts: ",totalRestart)
        print("Total number of moves taken: ",movesTotal)
        # Gives the total number of moves from starting point
        print("Number of moves in the solution set: ",movesSolution)
        # Gives number of steps in the solution set.
        i = 0
        print(notimes)
        notimes = notimes - 1
        while i < n:
            j = 0
            while j < n:
                if j == a[i]:
                    print(" Q ", end="")
                else:
                    print(" - ", end="")
                j += 1
            print()
            i += 1
    print('Success- '+str(success))
    print('Failure- '+str(failure))

if __name__ == '__main__':
    main()
```

3.2 Code for Sideways Move Hill Climbing

With Restart:

```
import random
import copy

class NQueenProblem(object):
    # This method calculates all the row conflicts for a queen placed in a
particular cell.
    @classmethod
    def rowHits(self, a, n):
        hitss = 0
        i = 0
        while i < n:
            j = 0
            while j < n:
                if i != j:
                    if a[i] == a[j]:
                        hitss += 1
                j += 1
            i += 1
        return hitss
```

```
    # This method calculates all the diagonal conflicts for a particular
position of the queen
    @classmethod
    def diagonalHits(self, a, n):
        hitss = 0
        d = 0
        i = 0
        while i < n:
            j = 0
            while j < n:
                if i != j:
                    d = abs(i - j)
                    if (a[i] == a[j] + d) or (a[i] == a[j] - d):
                        hitss += 1
                j += 1
            i += 1
        return hitss

    # This method returns total number of hitss for a particular queen
position
    @classmethod
    def totalHits(self, a, n):
        hitss = 0
        hitss = self.rowHits(a, n) + self.diagonalHits(a, n)
        return hitss

    # This method calculates the conflicts for the current state of the board
and quits whenever finds a better state.
    @classmethod
    def optimalSol(self, a, n):
        hitss = 0
        row = -1
        col = -1
        checkBetter = False
        best = []
        # Sets min variable to the hitss of current board so that if finds
better than this it will quit.
        mins = self.totalHits(a, n)
```

```
        best = copy.deepcopy(a)
        # Create a duplicate array for handling different operations
        i = 0
        while i < n:
            # This iteration is for each column
            if checkBetter:
                # If it finds and better state than the current, it will quit
                break
            m = best[i]
            j = 0
            while j < n:
                # This iteration is for each row in the selected column
                if j != m:
                    # This condition ensures that, current queen position is
not taken into consideration.
                    best[i] = j
                    # Assigning the queen to each position and then
calculating the hitss
                    hitss = self.totalHits(best, n)
                    if mins > hitss:
                        # If a better state is found, that particular column
and row values are stored
                        col = i
                        row = j
                        mins = hitss
                        checkBetter = True
                        break
                best[i] = m
                # Restoring the array to the current board position
                j += 1
            i += 1
        if col == -1 or row == -1:
            # If there is no better state found
            return False
        a[col] = row
        return True
        # Returns true to the main function if there is any better state
found
```

```
    # Below function generates a random state of the board
    @classmethod
    def randomGen(self, a, n):
        i = 0
        while( i < n):
            a[i] =random.randint(0,n-1) + 0
            i += 1


    # Below function verifies whether the current state of the board is the
solution(I.e with zero conflicts)
    @classmethod
    def checkSol(self, a, n):
        if self.totalHits(a, n) == 0:
            return True
        return False

    # Below method finds the solution for the n-queens problem with Min-
Conflicts algorithm
    @classmethod
    def minHits(self, b, n, iterations):
        # This array list is for storing the columns from which a random
column will be selected
        store = []
        self.fillList(store, n)
        randomCount = 0
        movesTotal = 0
        movesSolution = 0
        row = 0
        maxSteps = iterations
        # The maximum steps that can be allowed to find a solution with this
algorithm
        while not self.checkSol(b, n):
            # Loops until it finds a solution,
            randomSelection = random.randint(0,len(store)-1) + 0
            # Randomly selects a column from the available
            currentValue = b[store[randomSelection]]
```

```
            # This stores the current queue position in the randomly selected
column
            randomValue = store[randomSelection]
            mins = self.hitssminHits(b, n, randomValue)
            # Sets the minimum variable to the current queue hitss
            min_compare = mins
            while(not store):
                store.remove(randomSelection)
            i = 0
            while i < n:
                if currentValue != i:
                    b[randomValue] = i
                    col = self.hitssminHits(b, n, randomValue)
                    # Calculates the hitss of the queen at particular
position
                    if col < mins:
                        mins = col
                        row = i
                i += 1
            if min_compare == mins:
                # When there is no queen with minimum conflicts than the
current position
                if maxSteps != 0:
                    # Checks if the maximum steps is reached
                    if len(store) >= 0:
                        # checks whether there are columns available in the
Array List
                        b[randomValue] = currentValue
                        # restores the queen back to the previous position
                        maxSteps -= 1
                    else:
                        self.fillList(store, n)
                else:
                    # If the max steps is reached then, the board is
regenerated and initiated the max steps variable
                    randomCount += 1
                    movesSolution = 0
                    self.randomGen(b, n)
```

```
                    self.fillList(store, n)
                    maxSteps = iterations
            else:
                # When we find the the position in the column with minimum
conflicts
                movesTotal += 1
                movesSolution += 1
                b[randomValue] = row
                min_compare = mins
                store.clear()
                maxSteps -= 1
                self.fillList(store, n)
        print()
        i = 0
        while i < n:
            j = 0
            while j < n:
                if j == b[i]:
                    print(" Q ", end="")
                else:
                    print(" - ", end="")
                j += 1

            print()
            i += 1
        print("Total number of Random Restarts: ",randomCount)
        print("Total number of Moves: ", movesTotal)
        print("Number of Moves in the solution set: ", movesSolution)




    def minHitswithoutrestart(self, b, n, iterations):
        # This array list is for storing the columns from which a random
column will be selected
        store = []
        self.fillList(store, n)
        randomCount = 0
```

```
        movesTotal = 0
        movesSolution = 0
        global successmoves, failuremoves

        row = 0
        maxSteps = iterations
        # The maximum steps that can be allowed to find a solution with this
algorithm
        while not self.checkSol(b, n):
            # Loops until it finds a solution,
            randomSelection = random.randint(0,len(store)-1) + 0
            # Randomly selects a column from the available
            currentValue = b[store[randomSelection]]
            # This stores the current queue position in the randomly selected
column
            randomValue = store[randomSelection]
            mins = self.hitssminHits(b, n, randomValue)
            # Sets the minimum variable to the current queue hitss
            min_compare = mins
            while(not store):
                store.remove(randomSelection)
            i = 0
            while i < n:
                if currentValue != i:
                    b[randomValue] = i
                    col = self.hitssminHits(b, n, randomValue)
                    # Calculates the hitss of the queen at particular
position
                    if col < mins:
                        mins = col
                        row = i
                i += 1
            if min_compare == mins:
                # When there is no queen with minimum conflicts than the
current position
                if maxSteps != 0:

                    # Checks if the maximum steps is reached
```

```
                    if len(store) > 0:
                        # checks whether there are columns available in the
Array List
                        b[randomValue] = currentValue
                        # restores the queen back to the previous position
                        maxSteps -= 1
                    else:
                        self.fillList(store, n)
                else:
                    break
            else:
                # When we find the the position in the column with minimum
conflicts
                movesTotal += 1
                movesSolution += 1
                b[randomValue] = row
                min_compare = mins
                store.clear()
                maxSteps -= 1
                self.fillList(store, n)

        print()
        i = 0
        while i < n:
            j = 0
            while j < n:
                if j == b[i]:
                    print(" Q ", end="")
                else:
                    print("- ", end="")
                j += 1
            print()
            i += 1
        print("Total number of Random Restarts: ",randomCount)
        print("Total number of Moves: ", movesTotal)
        print("Number of Moves in the solution set: ", movesSolution)

        if queens.checkSol(b,n):
```

```
            successmoves+=movesTotal
        else:
            failuremoves+=movesTotal

    # Below function returns the hitss of a queen in a particular column of
the board
    @classmethod
    def hitssminHits(self, b, n, index):
        hitss = 0
        t = 0
        i = 0
        while i < n:
            if i != index:
                t = abs(index - i)
                if b[i] == b[index]:
                    hitss += 1
                elif b[index] == b[i] + t or b[index] == b[i] - t:
                    hitss += 1
            i += 1
        return hitss

    # Below function fills the Array List with numbers 0 to n-1
    @classmethod
    def fillList(self, store, n):
        i = 0
        while i < n:
            store.append(i)
            i += 1
        return

def main():

    n = int(input("Enter the number of Queens on the board: "))
    a = [None] * n
    b = [None] * n
    queens = NQueenProblem()
    queens.randomGen(a, n)
    b = copy.deepcopy(a)
```

```
    iterations = 0
    print()
    print("Please enter the maximum number of steps for iteration:")
    iterations = int(input("Please enter the value: "))
    queens.minHits(b, n, iterations)


    if(totalRestart == 0):
         print("Average Steps",movesTotal)
    else:
         print("Average restarts:",(movesTotal-1)/totalRestart)

if __name__ == '__main__':
    main()
```

Without Restart:

```
import random
import time
import copy

class NQueenProblem(object):
```

```
    # This method calculates all the row conflicts for a queen placed in a
particular cell.
    @classmethod
    def rowHits(self, a, n):
        hits = 0
        i = 0
        while i < n:
            j = 0
            while j < n:
                if i != j:
                    if a[i] == a[j]:
                        hits += 1
                j += 1
            i += 1
        return hits

    # This method calculates all the diagonal conflicts for a particular
position of the queen
    @classmethod
    def diagonalHits(self, a, n):
        hits = 0
        d = 0
        i = 0
        while i < n:
            j = 0
            while j < n:
                if i != j:
                    d = abs(i - j)
                    if (a[i] == a[j] + d) or (a[i] == a[j] - d):
                        hits += 1
                j += 1
            i += 1
        return hits

    # This method returns total number of hits for a particular queen
position
    @classmethod
    def totalHits(self, a, n):
```

```
        hits = 0
        hits = self.rowHits(a, n) + self.diagonalHits(a, n)
        return hits

    # This method calculates the conflicts for the current state of the board
and quits whenever finds a better state.
    #        Note: This function is used for Hill Climbing algorithm
    @classmethod
    def optimalSol(self, a, n):
 #       min = int()
        hits = 0
        row = -1
        col = -1
  #      m = int()
        checkBetter = False
        best = []
        # Sets min variable to the hits of current board so that if finds
better than this it will quit.
        mins = self.totalHits(a, n)
        best = copy.deepcopy(a)
        # Create a duplicate array for handling different operations
        i = 0
        while i < n:
            # This iteration is for each column
            if checkBetter:
                # If it finds and better state than the current, it will quit
                break
            m = best[i]
            j = 0

            while j < n:
                # This iteration is for each row in the selected column



                if j != m:
                    # This condition ensures that, current queen position is
not taken into consideration.
```

```
                    best[i] = j

                    self.printBoard(best,n)
                    p#rint('HI',best)
                    # Assigning the queen to each position and then
calculating the hits
                    hits = self.totalHits(best, n)
                    if mins > hits:
                        # If a better state is found, that particular column
and row values are stored
                        col = i
                        row = j
                        mins = hits
                        checkBetter = True
                        break
                best[i] = m
                # Restoring the array to the current board position
                j += 1

            i += 1
        if col == -1 or row == -1:
            # If there is no better state found
            print("Stuck at Local Maxima with " ,hits ,"collisions. Using
Random Restart...")
            return False
        a[col] = row
        return True
        # Returns true to the main function if there is any better state
found

    @classmethod
    def printBoard(self,b,n):
        i = 0
        while i < n:
            j = 0
            while j < n:
                if j == b[i]:
                    print(" Q ", end="")
```

```
                else:
                    print(" - ", end="")
                j += 1
            print()
            i += 1



    # Below function generates a random state of the board
    @classmethod
    def randomGen(self, a, n):
        i = 0
        while( i < n):
            a[i] =random.randint(0,n-1) + 0
            i += 1


    # Below function verifies whether the current state of the board is the
solution(I.e with zero conflicts)
    @classmethod
    def checkSol(self, a, n):
        if self.totalHits(a, n) == 0:
            return True
        return False

    # Below method finds the solution for the n-queens problem with Min-
Conflicts algorithm
    @classmethod
    def minHits(self, b, n, iterations):
        # This array list is for storing the columns from which a random
column will be selected
        store = []
        self.fillList(store, n)
        randomCount = 0
        movesTotal = 0
        movesSolution = 0
        row = 0
        maxSteps = iterations
```

```
        # The maximum steps that can be allowed to find a solution with this
algorithm
        while not self.checkSol(b, n):
            # Loops until it finds a solution,

            randomSelection = random.randint(0,len(store)-1) + 0
            # Randomly selects a column from the available
            currentValue = b[store[randomSelection]]
            # This stores the current queue position in the randomly selected
column
            randomValue = store[randomSelection]

            mins = self.hitsminHits(b, n, randomValue)
            # Sets the minimum variable to the current queue hits
            min_compare = mins
            while(not store):
                store.remove(randomSelection)

            i = 0
            while i < n:
                if currentValue != i:
                    b[randomValue] = i

                    col = self.hitsminHits(b, n, randomValue)
                    # Calculates the hits of the queen at particular position
                    if col < mins:
                        mins = col
                        row = i
                i += 1

            if min_compare == mins:
                # When there is no queen with minimum conflicts than the
current position
                if maxSteps != 0:
                    # Checks if the maximum steps is reached
                    if len(store) >= 0:
                        # checks whether there are columns available in the
Array List
```

```
                        b[randomValue] = currentValue

                        # restores the queen back to the previous position
                        maxSteps -= 1

                    else:
                        self.fillList(store, n)
                else:
                    break
                    # If the max steps is reached then, the board is
regenerated and initiated the max steps variable
                    randomCount += 1
                    movesSolution = 0
                    self.randomGen(b, n)
                    self.fillList(store, n)
                    maxSteps = iterations
            else:
                # When we find the the position in the column with minimum
conflicts
                movesTotal += 1
                movesSolution += 1
                b[randomValue] = row
                min_compare = mins
                store.clear()
                maxSteps -= 1
                self.fillList(store, n)
        print()
        i = 0

        while i < n:
            j = 0
            while j < n:
                if j == b[i]:
                    print(" Q ", end="")
                else:
                    print(" - ", end="")
                j += 1
```

```
            print()
            i += 1
        print("Total number of Moves: ", movesTotal)
        print("Number of Moves in the solution set: ", movesSolution)


    def minHitswithoutrestart(self, b, n, iterations):
        # This array list is for storing the columns from which a random
column will be selected
        store = []
        self.fillList(store, n)
        global succrate
        global failrate
        global movesTotal
        movesSolution = 0
        row = 0
        maxSteps = iterations
        # The maximum steps that can be allowed to find a solution with this
algorithm
        while not self.checkSol(b, n):
            # Loops until it finds a solution,
            randomSelection = random.randint(0,len(store)-1) + 0
            # Randomly selects a column from the available
            currentValue = b[store[randomSelection]]
            # This stores the current queue position in the randomly selected
column
            randomValue = store[randomSelection]
            mins = self.hitsminHits(b, n, randomValue)
            # Sets the minimum variable to the current queue hits
            min_compare = mins
    #        print("len store: ",len(store)-1,"randomSelection:
",randomSelection," currentValue: ",currentValue," randomValue:
",randomValue," mins: ",mins," min_compare: ",min_compare)
            while(not store):
                store.remove(randomSelection)
            i = 0
            while i < n:
                if currentValue != i:
```

```
                    b[randomValue] = i
                    col = self.hitsminHits(b, n, randomValue)
                    # Calculates the hits of the queen at particular position
                    if col < mins:
                        mins = col
                        row = i
                i += 1


            if min_compare == mins:
                # When there is no queen with minimum conflicts than the
current position
                if maxSteps != 0:
                    # Checks if the maximum steps is reached
                    if len(store) > 0:
                        # checks whether there are columns available in the
Array List
                        b[randomValue] = currentValue
                        # restores the queen back to the previous position
                        maxSteps -= 1

                    else:
                        self.fillList(store, n)
                else:
                    break
                    # If the max steps is reached then, the board is
regenerated and initiated the max steps variable
            else:
                # When we find the the position in the column with minimum
conflicts

                movesSolution += 1
                b[randomValue] = row
                min_compare = mins
                store.clear()
                maxSteps -= 1
                self.fillList(store, n)
```

```
        print()
        i = 0
        while i < n:
            j = 0
            while j < n:
                if j == b[i]:
                    print(" Q ", end="")
                else:
                    print("- ", end="")
                j += 1
            print()
            i += 1
        print(" totol moves ",movesTotal)
        print("Number of Moves in the solution set: ", movesSolution)

    def minHitswithoutrestart(self, b, n, iterations):
        # This array list is for storing the columns from which a random
column will be selected
        store = []
        self.fillList(store, n)
        randomCount = 0
        movesTotal = 0
        movesSolution = 0
        global successmoves, failuremoves

        row = 0
        maxSteps = iterations
        # The maximum steps that can be allowed to find a solution with this
algorithm
        while not self.checkSol(b, n):
            # Loops until it finds a solution,
            randomSelection = random.randint(0,len(store)-1) + 0
            # Randomly selects a column from the available
            currentValue = b[store[randomSelection]]
            # This stores the current queue position in the randomly selected
column
            randomValue = store[randomSelection]
            mins = self.hitsminHits(b, n, randomValue)
```

```
            # Sets the minimum variable to the current queue hits
            min_compare = mins
            while(not store):
                store.remove(randomSelection)
            i = 0
            while i < n:
                if currentValue != i:
                    b[randomValue] = i
                    col = self.hitsminHits(b, n, randomValue)
                    # Calculates the hits of the queen at particular position
                    if col < mins:
                        mins = col
                        row = i
                i += 1
            if min_compare == mins:
                # When there is no queen with minimum conflicts than the
current position
                if maxSteps != 0:

                    # Checks if the maximum steps is reached
                    if len(store) > 0:
                        # checks whether there are columns available in the
Array List
                        b[randomValue] = currentValue
                        # restores the queen back to the previous position
                        maxSteps -= 1
                    else:
                        self.fillList(store, n)
                else:
                    break
                    # If the max steps is reached then, the board is
regenerated and initiated the max steps variable

            else:
                # When we find the the position in the column with minimum
conflicts
                movesTotal += 1
                movesSolution += 1
```

```
                b[randomValue] = row
                min_compare = mins
                store.clear()
                maxSteps -= 1
                self.fillList(store, n)

        print()
        i = 0
        while i < n:
            j = 0
            while j < n:
                if j == b[i]:
                    print(" Q ", end="")
                else:
                    print("- ", end="")
                j += 1
            print()
            i += 1
        print("Total number of Moves: ", movesTotal)
        print("Number of Moves in the solution set: ", movesSolution)

        if queens.checkSol(b,n):
            successmoves+=movesTotal
        else:
            failuremoves+=movesTotal

    # Below function returns the hits of a queen in a particular column of
the board
    @classmethod
    def hitsminHits(self, b, n, index):
        hits = 0
        global c
        t = 0
        i = 0
        while i < n:
            if i != index:
                t = abs(index - i)
                if b[i] == b[index]:
```

```
                    hits += 1
                elif b[index] == b[i] + t or b[index] == b[i] - t:
                    hits += 1
            i += 1

            self.printBoard(b,n)
            print('Queen Position',b)
            c+=1
        return hits

    # Below function fills the Array List with numbers 0 to n-1
    @classmethod
    def fillList(self, store, n):
        i = 0
        while i < n:
            store.append(i)
            i += 1
        return
executeiter = 0
totalRestart = 0
movesTotal = 0
movesSolution = 0
succrate = 0
failrate = 0



def main():
    success=0
    failure=0
    c=0
    successmoves=0
    failuremoves=0
    n = int(input("Please enter the number of queens on the board: "))

    a = [None] * n
    b = [None] * n
    queens = NQueenProblem()
```

```
    queens.randomGen(a, n)
        # Randomly generate the board
    b = copy.deepcopy(a)
    print("How many times do you want to run the code?")
    notimes = int(input("Please enter the value:"))
    print("Please enter the maximum number of steps for iteration:")
    iterations = int(input("Please enter the value:"))
    while (notimes!=0):
        startTime = time.time()
        queens.minHitswithoutrestart(b, n, iterations)

        movesTotal += 1
        if queens.checkSol(b,n):

            print('Yes')
            success+=1
        else:
            print('No')
            failure+=1
        endTime = time.time()
        print("Time Taken in milli seconds: ",(endTime - startTime))
        print(notimes)
        notimes = notimes - 1
        queens.randomGen(b,n)
    if(failuremoves == 0):
        print(" average success steps",successmoves)
    else:
        print(" average failure steps",failuremoves)

    print('Success- '+str(success))
    print('Failure- '+str(failure))
    print('Successmoves- '+str(successmoves))
    print('Failuremoves- '+str(failuremoves))
    print("moves",c)

if __name__ == '__main__':
    main()
```

SAMPLE OUTPUT

Please select one from the below options:
1. Solve n queens with **Hill Climbing Steepest Ascent** without restart
2. exit

Please enter the choice:1
Please enter the value of n:4
Please enter the value of iterations:1
**********Hill Climbing without Random Restart*********
Q - - -
- Q - -
- - Q -
- - - Q
Queen Position [0, 1, 2, 3]
- Q - -
- Q - -
- - Q -
- - - Q
Queen Position [1, 1, 2, 3]
- - Q -
- Q - -
- - Q -
- - - Q
Queen Position [2, 1, 2, 3]
- - - Q
Q - - -
- - Q -
- - - Q
Queen Position [3, 0, 2, 3]
solution not found

Total number of moves taken: 4
Number of moves in the solution set: 1
Time Taken in milli seconds: 0.0010073184967041016
 - - - Q
 Q - - -
 - - Q -
 - - - Q

Please select one from the below options:
1. Solve n queens with **Hill Climbing Steepest Ascent with Random Restart**
2.exit

enter your choice:1

Please enter the value of n:4
**********Hill Climbing with Random Restart*********
Reached Local Maxima with 8 Regenerating randomly
Number of Restarts: 1
Total number of moves taken: 2
Number of moves in the solution set: 2
Time Taken in milli seconds: 0.0
 - - Q -
 Q - - -
 - - - Q
 - Q - -
 average steps 3.0

Please select one from the below options:
1. Min Conflict method with random restart
2.exit

Please enter the choice:1

Please enter the value of n:4
**********Sideways with Random Restart*********
Reached Local Maxima with 4 Regenerating randomly
Number of Restarts: 1
Total number of moves taken: 20

Number of moves in the solution set: 20
Time Taken in milli seconds: 0.000982046127319336
 - - Q -
 Q - - -
 - - - Q
 - Q - -
 average steps 20.0

Please select one from the below options:
1. Min Conflict method without random restart
2.exit

Please enter the choice:1

Please enter the value of n:4
*******Min Conflict Sideways Without Random Restart*******
How many times do you wanna run the code??

Please enter the value:1
Please enter the maximum number of steps for iteration:

Please enter the value:25
 - Q - -
 - Q - -
 Q - - -
 - - Q -
Queen Position [1, 1, 0, 2] 1
 - Q - -
 - Q - -
 Q - - -
 - - Q -
Queen Position [1, 1, 0, 2] 2
 - Q - -
 - Q - -
 Q - - -
 - - Q -
Queen Position [1, 1, 0, 2] 3
 - Q - -
 - Q - -

Q - - -
- - Q -
Queen Position [1, 1, 0, 2] 4
- Q - -
Q - - -
Q - - -
- - Q -
Queen Position [1, 0, 0, 2] 1
- Q - -
Q - - -
Q - - -
- - Q -
Queen Position [1, 0, 0, 2] 2
- Q - -
Q - - -
Q - - -
- - Q -
Queen Position [1, 0, 0, 2] 3
- Q - -
Q - - -
Q - - -
- - Q -
Queen Position [1, 0, 0, 2] 4
- Q - -
- - Q -
Q - - -
- - Q -
Queen Position [1, 2, 0, 2] 1
- Q - -
- - Q -
Q - - -
- - Q -
Queen Position [1, 2, 0, 2] 2
- Q - -
- - Q -
Q - - -
- - Q -
Queen Position [1, 2, 0, 2] 3
- Q - -
- - Q -

```
 Q  -  -  -
 -  -  Q  -
Queen Position [1, 2, 0, 2] 4
 -  Q  -  -
 -  -  -  Q
 Q  -  -  -
 -  -  Q  -
Queen Position [1, 3, 0, 2] 1
 -  Q  -  -
 -  -  -  Q
 Q  -  -  -
 -  -  Q  -
Queen Position [1, 3, 0, 2] 2
 -  Q  -  -
 -  -  -  Q
 Q  -  -  -
 -  -  Q  -
Queen Position [1, 3, 0, 2] 3
 -  Q  -  -
 -  -  -  Q
 Q  -  -  -
 -  -  Q  -
Queen Position [1, 3, 0, 2] 4

-  Q - -
- - -  Q
 Q - - -
- -  Q -
Total number of Random Restarts:  0
Total number of Moves:  1
Number of Moves in the solution set:  1
Yes
Time Taken in milli seconds:  0.007997751235961914
1
 average success steps 1
Success- 1
Failure- 0
Successmoves- 1
Failuremoves- 0
```

PROGRAM IMPLEMENTAION

5.1 Implement steepest-ascent hill climbing

a. Run several times, say 100 to 500, and report success and failure rates
We did run the 8 queens problem 100 times, it was success for 15 times and failed **85 times**
b. The average number of steps when it succeeds
It did succeed in 3 steps
c. The average number of steps when it fails
It did fail in 4 steps
d. The search sequences from three random initial configurations
Case 1:

Please enter the number of Queens on board: 8

Please enter the value of iterations:100

Q - - - - - - -	- - Q - - - - -	- - - Q - - - -	
- - - - - - - Q	- - - - - - - Q	- - - - - - - Q	
Q - - - - - - -	Q - - - - - - -	Q - - - - - - -	
- - - - - - - Q	- - - - - - - Q	- - - - - - - Q	
- - - - - Q - -	- - - - - Q - -	- - - - - Q - -	
- Q - - - - - -	- Q - - - - - -	- Q - - - - - -	
- - Q - - - - -	- - Q - - - - -	- - Q - - - - -	
- - - - - - - Q	- - - - - - - Q	- - - - - - - Q	

Solution Not Found

Total number of moves taken: 3

Number of moves in the solution set: 1

Case 2:

Q - - - - - - - - Q - - - - - - - - - Q - - - - Q - - - - - - - - - - - - Q - - - - - - - - - Q - - - Q - - - - - - - - - Q - - -	- Q - - - - - - - Q - - - - - - - - - Q - - - - Q - - - - - - - - - - - - Q - - - - - - - - - Q - - - Q - - - - - - - - - Q - - -	- - - Q - - - - - Q - - - - - - - - - Q - - - - Q - - - - - - - - - - - - Q - - - - - - - - - Q - - - Q - - - - - - - - - Q - - -	- - - - Q - - - - Q - - - - - - - - - Q - - - - Q - - - - - - - - - - - - Q - - - - - - - - - Q - - - Q - - - - - - - - - Q - - -

Solution Not Found

Total number of moves taken: 4

Number of moves in the solution set: 1

Case 3:

Q - - - - - - - - - - - - - - Q Q - - - - - - - - - - - - - - Q - - - - - Q - -	- - Q - - - - - - - - - - - - Q Q - - - - - - - - - - - - - - Q - - - - - Q - -	- - - Q - - - - - - - - - - - Q Q - - - - - - - - - - - - - - Q - - - - - Q - -	

- Q - - - - - - - - Q - - - - - - - - - - - - Q	- Q - - - - - - - - Q - - - - - - - - - - - - Q	- Q - - - - - - - - Q - - - - - - - - - - - - Q	

Solution Not Found

Total number of moves taken: 3

Number of moves in the solution set: 1

5.2 Hill-climbing search with sideways move

a. Run several times, say 100 to 500, and report success and failure rates

We did run the 8 queens problem 100 times, it was success for 93 times and failed for 7 times

b. The average number of steps when it succeeds

Average steps is 20 to success

c. The average number of steps when it fails

Average steps is 62 to failure

d. The search sequences from three random initial configurations

Case 1: (Failure – 63 steps)

Please enter the number of Queens on board: 8

How many times do you want run the code?
Please enter the value:100
Please enter the maximum number of steps for iteration, limit on sideways moves to avoid infinite loop:
Please enter the value:20

- Q - - - - - - - - - - - - - Q - - - - - - - Q - Q - - - - - - Q - - - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -	- Q - - - - - - - - - - - - - Q - - - - - - - Q - Q - - - - - - - Q - - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -	- Q - - - - - - - - - - - - - Q - - - - - - - Q - Q - - - - - - - Q - - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -	- Q - - - - - - - - - - - - - Q - - - - - - - Q - Q - - - - - - - - - Q - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -
- Q - - - - - - - - - - - - - Q - - - - - - - Q - Q - - - - - - - - - - Q - - -	- Q - - - - - - - - - - - - - Q - - - - - - - Q - Q - - - - - - - - - - - Q - -	- Q - - - - - - - - - - - - - Q - - - - - - - Q - Q - - - - - - - - - - - - Q -	- Q - - - - - - - - - - - - - Q - - - - - - - Q - Q - - - - - - - - - - - - - Q

- - - - - Q - - - - - Q - - - - - - - - - - Q -	- - - - - Q - - - - - Q - - - - - - - - - - Q -	- - - - - Q - - - - - Q - - - - - - - - - - Q -	- - - - - Q - - - - - Q - - - - - - - - - - Q -
- Q - - - - - - - - - - - - - Q - - - - - - - Q - Q - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - Q - - - - - - -	- Q - - - - - - - - - - - - - Q - - - - - - - Q - Q - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - Q - - - - - -	- Q - - - - - - - - - - - - - Q - - - - - - - Q - Q - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - Q - - - - -	- Q - - - - - - - - - - - - - Q - - - - - - - Q - Q - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - Q - - - -
- Q - - - - - - - - - - - - - Q - - - - - - - Q - Q - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - Q - - -	- Q - - - - - - - - - - - - - Q - - - - - - - Q - Q - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - Q - -	- Q - - - - - - - - - - - - - Q - - - - - - - Q - Q - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - Q - -	- Q - - - - - - - - - - - - - Q - - - - - - - Q - Q - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - - Q
- Q - - - - - - - - - - - - - Q - - - - - - - Q Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -	- Q - - - - - - - - - - - - - Q - - - - - - - Q Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -	- Q - - - - - - - - - - - - - Q - - - - - - - Q - - Q - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -	- Q - - - - - - - - - - - - - Q - - - - - - - Q - - - Q - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -
- Q - - - - - - - - - - - - - Q - - - - - - - Q - - - - Q - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -	- Q - - - - - - - - - - - - - Q - - - - - - - Q - - - - - Q - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -	- Q - - - - - - - - - - - - - Q - - - - - - - Q - - - - - - Q - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -	- Q - - - - - - - - - - - - - Q - - - - - - - Q - - - - - - - Q - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -
- Q - - - - - - - - - - - - - Q - - - - - - - Q Q - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q - - - - - - - - - - Q -	- Q - - - - - - - - - - - - - Q - - - - - - - Q Q - - - - - - - - - Q - - - - - - Q - - - - - - - - - Q - - - - - - - - - - Q -	- - - - Q - - - - - - - - - - Q - - - - Q - - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -	- - - - - Q - - - - - - - - - Q - - - - Q - - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -
- - - - - - Q - - - - - - - - Q - - - - Q - - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - -	- - - - - - - Q - - - - - - - Q - - - - Q - - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - -	- Q - - - - - - - - - - - - - Q Q - - - - - - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - -	- Q - - - - - - - - - - - - - Q - Q - - - - - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - -

- - - - - - Q - 	- - - - - - Q - 	- - - - - - Q - 	- - - - - - Q -
- Q - - - - - - - - - - - - - Q - - Q - - - - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -	- Q - - - - - - - - - - - - - Q - - - Q - - - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -	- Q - - - - - - - - - - - - - Q - - - Q - - - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -	- Q - - - - - - - - - - - - - Q - - - - - Q - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -
- Q - - - - - - - - - - - - - Q - - - - - - Q - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -	- Q - - - - - - - - - - - - - Q - - - - - - - Q Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -	- Q - - - - - - - - - - - - - Q - - - - Q - - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - Q - - - - - - -	- Q - - - - - - - - - - - - - Q - - - - Q - - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - Q - - - - - -
- Q - - - - - - - - - - - - - Q - - - - Q - - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - Q - - - - -	- Q - - - - - - - - - - - - - Q - - - - Q - - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - Q - - - -	- Q - - - - - - - - - - - - - Q - - - - Q - - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - Q - - -	- Q - - - - - - - - - - - - - Q - - - - Q - - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - Q - -
- Q - - - - - - - - - - - - - Q - - - - Q - - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - Q - -	- Q - - - - - - - - - - - - - Q - - - - Q - - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - - Q	- Q - - - - - - - - - - - - - Q Q - - - - - - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -	- Q - - - - - - - - - - - - - Q - Q - - - - - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -
- Q - - - - - - - - - - - - - Q - - Q - - - - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -	- Q - - - - - - - - - - - - - Q - - - Q - - - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -	- Q - - - - - - - - - - - - - Q - - - Q - - - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -	- Q - - - - - - - - - - - - - Q - - - Q - - - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -
- Q - - - - - - - - - - - - - Q - - - - - Q - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -	- Q - - - - - - - - - - - - - Q - - - - - - Q - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -	- Q - - - - - - - - - - - - - Q - - - - - - - Q Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -	- Q - - - - - - - - - - - - Q - - - - - Q - - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -

- Q - - - - - - - - - - - - Q - - - - - Q - - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - - - - - Q -			

Case 2: (Success- 24 steps)

- - - - - Q - - - - - - - - - Q Q - - - - - - - - - - Q - - - - Q - - - - - - - - - Q - - - - - - - Q - - - - - - - - - - - - Q	- - - - - Q - - - - - - - - - Q Q - - - - - - - - - - Q - - - - Q - - - - - - - - - Q - - - - - - - Q - - - - - - - - - - - - Q	- - - - - Q - - - - - - - - - Q Q - - - - - - - - - - Q - - - - Q - - - - - - - - - Q - - - - - - - Q - - - - - - - - - - - - Q	- - - - - Q - - - - - - - - - Q - Q - - - - - - - - - Q - - - - Q - - - - - - - - - Q - - - - - - - Q - - - - - - - - - - - - Q
- - - - - Q - - - - - - - - - Q - Q - - - - - - - - - Q - - - - Q - - - - - - - - - Q - - - - - - - Q - - - - - - - - - - - - Q	- - - - - Q - - - - - - - - - Q - - Q - - - - - - - - Q - - - - Q - - - - - - - - - Q - - - - - - - Q - - - - - - - - - - - - Q	- - - - - Q - - - - - - - - - Q - - - - Q - - - - - - Q - - - - Q - - - - - - - - - Q - - - - - - - Q - - - - - - - - - - - - Q	- - - - - Q - - - - - - - - - Q - - - - - Q - - - - - Q - - - - Q - - - - - - - - - Q - - - - - - - Q - - - - - - - - - - - - Q
- - - - - Q - - - - - - - - - Q - - - - Q - - - - - - Q - - - - Q - - - - - - - - - Q - - - - - - - - - Q - - - - - - - Q - - -	- - - - - Q - - - - - - - - - Q - - - - - - - Q - - - Q - - - - Q - - - - - - - - - Q - - - - - - - - - Q - - - - - - - Q - - -	- - - - - Q - - - - - - - - - Q - Q - - - - - - - Q - - - - - - Q - - - - - - - - - Q - - - - - - - - - Q - - - - - - - Q - - -	- - - - - Q - - - - - - - - - Q - Q - - - - - - - - - Q - - - - Q - - - - - - - - - Q - - - - - - - - - Q - - - - - - - - - Q -
- - - - - Q - - - - - - - - - Q - Q - - - - - - - - - Q - - - - Q - - - - - - - - - Q - - - - - - - - - Q - - - - - - - - - - Q	- - - - - Q - - - - - - - - - Q - Q - - - - - - - - - Q - - - - Q - - - - - - - Q - - - - - - - - - - - Q - - - - - Q - - - - -	- - - - - Q - - - - - - - - - Q - Q - - - - - - - - - Q - - - - Q - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - -	- - - - - Q - - - - - - - - - Q - Q - - - - - - - - - Q - - - - Q - - - - - - - - - - Q - - - - - - - - Q - - - - - Q - - - - -
- - - - - Q - - - - - - - - - Q - Q - - - - - - - - - Q - - - -	- - - - - Q - - - - - - - - - Q - Q - - - - - - - - - Q - - - -	- - - - - Q - - - - - - - - - Q - Q - - - - - - - - - Q - - - -	----- Q - - ------- Q - Q - - - - - - --- Q - - - -

Q - - - - - - - - - - - Q - - - - - - - Q - - - - - Q - - - - -	Q - - - - - - - - - - - - - Q - - - - - Q - - - - - Q - - - - -	Q - - - - - - - - - - - - - - Q - - - - Q - - - - - Q - - - - -	Q - - - - - - - - - - - - - Q - - - - - Q - - - - - Q - - - - -

Case 3: (Success – 24 steps)

Q - - - - - - - Q - - - - - - - - - - - - Q - - - - - Q - - - - - - Q - - - - - Q - - - - - - - - - - Q - - - - - - - - Q - - -	- - - - Q - - - Q - - - - - - - - - - - - Q - - - - - Q - - - - - - Q - - - - - Q - - - - - - - - - - Q - - - - - - - - Q - - -	- - - - - - - Q Q - - - - - - - - - - - - Q - - - - - Q - - - - - - Q - - - - - Q - - - - - - - - - - Q - - - - - - - - Q - - -	Q - - - - - - - Q - - - - - - - - - - - - Q - - - - - Q - - - - - - Q - - - - - Q - - - - - - - - - - Q - - - - - - - - Q - - -
- - - Q - - - - Q - - - - - - - - - - - - Q - - - - - Q - - - - - - Q - - - - - Q - - - - - - - - - - Q - - - - - - - - Q - - -	- - - - - - Q - Q - - - - - - - - - - - - Q - - - - - Q - - - - - - Q - - - - - Q - - - - - - - - - - Q - - - - - - - - Q - - -	- Q - - - - - - - Q - - - - - - - - - - - Q - - - - - Q - - - - - - Q - - - - - Q - - - - - - - - - - Q - - - - - - - - Q - - -	- Q - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - - - Q - - - - - Q - - - - - - - - - - Q - - - - - - - - Q - - -
- Q - - - - - - - - - - Q - - - - - - - - Q - - - - - Q - - - - - - Q - - - - - Q - - - - - - - - - - Q - - - - - - - - Q - - -	- Q - - - - - - - - - - - - - Q - - - - - Q - - - - - Q - - - - - - Q - - - - - Q - - - - - - - - - - Q - - - - - - - - Q - - -	- Q - - - - - - - - - - - - - Q - - - - - Q - - - - - Q - - - - - - Q - - - - - Q - - - - - - - Q - - - - - - - - - - - Q - - -	- Q - - - - - - - - - - - - - Q - - - - - Q - - - - - Q - - - - - - Q - - - - - Q - - - - - - - - - Q - - - - - - - - - Q - - -
- - - - - Q - - - - - - - - - Q - Q - - - - - - - - - Q - - - - Q - - - - - - - - - Q - - - - - - - - - Q - - - - - - - - - - Q	- - - - - Q - - - - - - - - - Q - Q - - - - - - - - - Q - - - - Q - - - - - - - Q - - - - - - - - - - - Q - - - - - Q - - - - -	- - - - - Q - - - - - - - - - Q - Q - - - - - - - - - Q - - - - Q - - - - - - - - Q - - - - - - - - - - Q - - - - - Q - - - - -	- - - - - Q - - - - - - - - - Q - Q - - - - - - - - - Q - - - - Q - - - - - - - - - - Q - - - - - - - - Q - - - - - Q - - - - -
- Q - - - - - - - - - - - - - Q - - - - - Q - - - - - Q - - - - - - Q - - - - - Q - - - - - - - - - - Q - - - - - - - - Q - - -	- Q - - - - - - - - - - - - - Q - - - - - Q - - - - - Q - - - - - - Q - - - - - Q - - - - - - - - - - Q - - - - - - - - Q - - -	- Q - - - - - - - - - - - - - Q - - - - - Q - - - - - Q - - - - - - Q - - - - - Q - - - - - - - Q - - - - - - - - - - - Q - - -	- Q - - - - - - - - - - - - - Q - - - - - Q - - - - - Q - - - - - - Q - - - - - Q - - - - - - - - - Q - - - - - - - - - Q - - -

Q - - - - - - - - - Q - - - - - - - - - Q - - -	Q - - - - - - - - - - - - - - Q - - - - - Q - - -	Q - - - - - - - - - - - - - - Q - - - - - Q - - -	- - - - - - - Q - - Q - - - - - Q - - - - - - - - - - - - - Q - - - - - Q - - -

5.3 Random-restart hill-climbing search

a. The average number of random restarts used without sideways move
Average number of restarts is 7
b. The average number of steps required without sideways move
Average number of steps is 22
c. The average number of random restarts used with sideways move
Average number of restarts is 2
d. The average number of steps required with sideways move
Average number of steps is 20

CONCLUSION

In this project we see that when n queens are implemented using Steepest-ascent hill climbing method, only 14% of the time it solves the problem where as 86% of the time it gets stuck at a local minimum. However, it takes only 4 steps on average when it succeeds and 3 on average when it gets stuck – (for a state space with 8^8 =~17 million states)
Unfortunately, hill climbing often gets stuck for the following reasons:
– Local maxima: This figure is a local maximum (i.e., a local minimum for the cost h =1); every move of a single queen makes the situation worse.
– Ridges: a ridge is shown in the Figure. Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
– Plateau: a plateau is a flat area of the state-space landscape.
• It can be a flat local maximum, from which no uphill exit exists, or
• a shoulder, from which progress is possible. A hill-climbing search might get lost on the plateau.

In order to escape the shoulders, we implement n queens using Sideways move. For 8-queens, we allow sideways moves with a limit of 100 which raises the percentage of problem instances solved from 14 to 94%. However, it takes 21 steps for every successful solution and 64 steps for each failure.

Both the methods – Steepest-ascent and Sideways move Hill climbing are incomplete -- they often fail to find a goal when one exists because they can get stuck on local maxima. Therefore, we implement the n queen problem using Random restart Hill climbing method. The algorithm conducts a series of hill-climbing searches from randomly generated initial states, until a goal is found. It is complete with probability approaching 1, because it will eventually generate a goal state as the initial state.

REFERENCES

[1] Hill Climbing, Local Search, Wikipedia. Accessed: October 2018.

[2] N Queens Problem, Geeks for Geeks Accessed: October 2018.

[3] Topic: n-queens, GitHub Accessed: October 2018.