

Project report
on
**Solving N-Queens problem using Hill-
Climbing Algorithm and its variants**

Project Guidance By

Dr. Dewan Ahmed

Team details

Aditya Kadimi

akadimi@uncc.edu

801167778

Mrudula Ravipati

mravipat@uncc.edu

801167901

Sai Harish Paleti

spaleti@uncc.edu

801167778.

Date: 03/27/2020

AIM

To solve n-queens problem using hill-climbing search and its variants.

PROBLEM STATEMENT

Implement Hill-climbing search, Hill-climbing search with sideway moves and Random-restart hill-climbing with and without sideways move and apply it to n-queens problem. List average number of steps when the algorithm succeeds and fails along with the success and failure rate for multiple iterations.

N-QUEENS PROBLEM

The N-queens puzzle is the problem of placing N queens on a N x N chessboard such that no two queens attack each other. The queen is the most powerful piece in chess and can attack from any distance horizontally, vertically, or diagonally. Thus, a solution requires that no two queens share the same row, column, or diagonal.

PROBLEM FORMULATION

Initial State: A random arrangement on n queens, with one in each column.

Goal State: N queens placed on the board such that no two queens can attack each other.

States: Any arrangement of n queens, one in each column.

Actions: Move any attacked queen to another square in the same column.

Performance: Number of steps and success rate to find a solution.

HILL-CLIMBING ALGORITHM

Hill Climbing is heuristic search used for mathematical optimization problems in the field of Artificial Intelligence. It is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by making an incremental change to the solution. If the change produces a better solution, another incremental change is made to the new solution, and so on until no further improvements can be found.

Steepest-Ascent Hill-climbing: It first examines all the neighboring nodes and then selects the node closest to the solution state as next node with best heuristic value. If no best successor is found then the search fails.

$$f(n) = g(n) + h(n)$$

$g(n)$ = cost so far to reach n

$h(n)$ = estimated cost from n to goal

$f(n)$ = estimated total cost of path through n to goal

Heuristic Functions

The heuristic function is a way to inform the search regarding the direction to a goal. It provides an information to estimate which neighboring node will lead to the goal. The two heuristic functions that we considered for solving 8-puzzle

problem are

Misplaced Tile

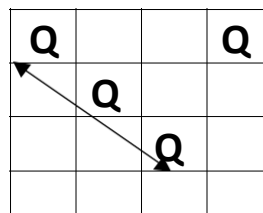
The number of misplaced tiles calculated by comparing the current state and goal state.

Manhattan Distance

The distance between two tiles measured along the axes of right angles. It is the sum of absolute values of differences between goal state (i, j) coordinates and current state (l, m) coordinates respectively, i.e. $|i - l| + |j - m|$

HEURISTIC FUNCTION:

The Heuristic function in the N queen problem is the number of pairs of queens that are attacking each other. The best successor is the state with low heuristic value.



The heuristic value for the above problem is four since there are four pairs of queens that are attacking each other at this moment.

N- Queens Puzzle

For solution searching, it would be most useful to distil the possible arrangements of tiles as individual States. Thus, each State shows a possible combination of tile positions within the given puzzle space. The collection of all possible States is called the State Space. With the increase of N or M of the puzzle, the size of the State Space shall increase exponentially.

In every state, the empty space position determines which States can be transitioned to. For instance, when the empty space is in the middle of a 3x3 puzzle, tiles at the Top, Bottom, Left or Right can move into it. But if the empty space is at the top left corner, only the right or bottom tiles can slide into it.

Thus, after each slide, a new State is transitioned into. If puzzle is to begin with an Initial State of tile arrangements, then its subsequent transitions into other States can be represented by a Graph. A search attempt will need to begin with an Initial State and a Goal State to achieve. As puzzle traversal can often pass through the same state at different intervals. We will consider the instances of decisions as nodes. By aligning the node arrangements to start from the Initial Node to possible routes leading to the Goal nodes, a search tree is formed.

Sl No	Environment Characteristics of Puzzle	
		Description
1.	Performance	Arrangement of tiles/cells/blocks in the whole puzzle space. Main performance gauge is from the least number of moves to solve the puzzle.
2.	Environment	Puzzle space determined by N (columns) and M (rows), always with a single empty space for tiles to slide into. Numbers range from 1 to (N*M)-1. Initial state arrangements must be derived from Goal state arrangement or else there will not be possible solutions.
3.	Actuators	Tiles are moved into the empty space, either from Top, Bottom, Left or Right of the empty space.
4.	Sensors	Fully software, so the agent will have full view of the puzzle space.

Hill Climbing Algorithm

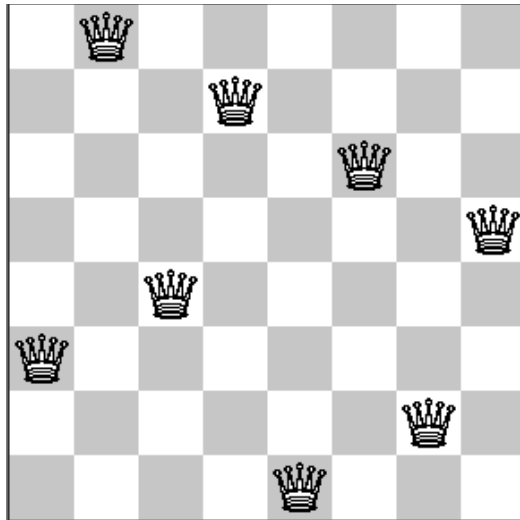
Hill Climbing works disregards memory of explored nodes. Therefore, it travels down the Search Tree by selecting the successor with the cheapest heuristics value, without retaining memory of explored states. This will ensure that the heuristics technique functions with minimal use of memory, least computation possible but still retain the advantage of an informed method of solution finding. The downside of Hill Climbing is that due to the absence of memory, resulting in the possibility of repeating the same states and getting stuck in some state of local maxima.

Hill Climbing for 8 queens Problem

Hill climbing search for this 8-Queen puzzle, in order to reach the goal state where $h = 0$, it will continue to loop to find moves in the direction of decreasing $h(n)$. It will terminate when there is no lower $h(n)$ than the previous ones. Hill climbing search will randomly generate 8 random placement of the queen in the 8x8 board after the initial state it will then calculate the $h(n)$ and then during the next state it will swap the Q position column by column in search of the $h(n)$ that is lesser than the previous $h(n)$ until it reach $h=0$. Hence, based on the evaluation function $f(n)=h(n)$, so the results will be $f(n)1=0$. The board will terminate if there is no $h(n)$ that is less than the previous $h(n)$. When board is clear, a new random placement of the queens is placed again and the process is repeated until it reaches the goal state.

N-Queens: Steepest Hill Climbing:

The n-queens problem was first invented in the mid-1800s as a puzzle for people to solve in their spare time, but now serves as a good tool for discussing computer search algorithms. In chess, a queen is the only piece that can attack in any direction. The puzzle is to place a number of queens on a board in such a way that no queen is attacking any other. For example:



The N-queens problem is the problem of placing 'n' chess queens on an $n \times n$ chessboard so that no two queens threaten each other. This means that no two queens can be in same row, column or diagonal. We can find solutions for all natural numbers 'n' except for $n=2$ and $n=3$. Here the problem is solved using a complete-state formulation, which means we start with all 8 queens on the board and move them around to reach the goal state. We represent the $n \times n$ chess board as a matrix.

The classic combinatorial problem is to place N-Queens on a chessboard so that no two attack each other. In the chess Queens attacking in three directions i.e. horizontally, vertically and diagonally. The problem can be generalized as placing 'n' non attacking queens on an $N \times N$ chessboard. Since each queen must be on a different row and column, we can assume that queen "i" is placed in ith column. All solutions to the NQP can therefore be represented as n-tuples (q_1, q_2, \dots, q_n) that are permutations of an n-tuple $(1, 2, 3, \dots, n)$. Position of a number in the tuple represents queen's column position, while its value represents queen's row position (counting from the bottom) using this representation, the solution space where two of the constraints (row and column conflicts) are already satisfied should be searched in order to eliminate the diagonal conflicts. Complexity of this problem is $O(n!)$. The N-Queens problem is a generalization of the 8-Queens problem posed by a German chess player, Max Bezzel in 1848. The objective of the N-Queens problem is to arrange N-Queens so that no queen may attack each queen. Thus each column, row, diagonal, and anti-diagonal must contain one and only one queen.

PROCESS OF SOLVING N-QUEENS

- Suppose you have 8 chess Queens and chess board of size 8×8 .
- Queens can be placed on the chess board so no two queens are attacking each other.
- Two Queens are not allowed in the same column.
- Two Queens are not allowed in the same column, in the same row.
- Two Queens are not allowed in the same column, in the same row, or along the same diagonal.
- The number of Queens and the size of the board can differ.
- It looks like hard to generate one valid placement.

- The program uses a stack to keep track of where each Queens is placed.
- Each time the program decides to place a Queens on the board, the position of the new Queens is stored in a record which is placed in the stack.
- We also have an integer variable to keep track of how many rows have been filled so far.
- Each time we try to place a new Queens in the next row, we start by placing the Queens in the first column.
- If there is a clash with another Queens, then we shift the new Queens to the next column.
- If another clash occurs, the Queens is shifted rightward again.
- When there are no clash, we stop and add one to the value of filled.

Program Design and Code Explanation

Screenshots:

Enter the the value for number of queens in n-Queen problem: 8

Please enter the Runtime: 500

Enter the the value for number of queens in n-Queen problem: 8

Please enter the Runtime: 500

First Path for Steepest Ascent

####Q###

Q#####Q

##Q##Q##

#####

#####

#####

#Q####Q#

###Q####

####Q###

Q#####Q

#####Q##

#####

##Q#####

#####

#Q####Q#

###Q####

####Q###

#####Q

#####Q##

Q#####

##Q#####

#####

#Q####Q#

###Q####

#Q##Q###

#####Q

#####Q##

Q#####

##Q#####

#####

#####Q#

###Q####

#Q#####

#####Q

#####Q##

Q#####

##Q#####

####Q###

#####Q#

###Q####

Path cost: 5

Second Path for Steepest Ascent

#####Q

Q#####

####Q###

###Q####

#####

#QQ##QQ#

#####

#####

#####Q

Q#####

####Q###

###Q####

#####

##Q##QQ#

#####

#Q#####

#####Q

Q#####

####Q###

###Q####

#####

#####QQ#

##Q#####

#Q#####

#####Q

Q#####

####Q###

###Q####

#Q#####

#####QQ#

##Q#####

#####

#####Q

Q#####

####Q###

###Q####

#Q#####

#####Q#

##Q#####

#####Q##

Path cost: 5

Third Path for Steepest Ascent

#####Q#

#Q##Q###

#####Q

#####Q##

Q#####

##Q#####

###Q####

#####

#####Q#

#Q##Q###

#####Q

#####Q##

Q#####

##Q#####

#####

###Q####

Path cost: 2

First Path for Steepest Ascent Sideways Move

###QQ###

#####Q

#####Q#

#####

Q#####

#####

##Q#####

#Q###Q##

###QQ###

#####Q

#####Q#

##Q#####

Q#####

#####

#####

#Q###Q##

###QQ###

#####Q

#####Q#

##Q#####

Q#####

#####

#Q#####

#####Q##

###Q####

#####Q

####Q#Q#

##Q#####

Q#####

#####

#Q#####

#####Q##

###Q####

#####Q

####Q###

##Q#####

Q#####

#####Q#

#Q#####

#####Q##

Path cost: 5

Second Path for Sideways Move

#####Q#

#####

##Q#####

#Q###Q#Q

#####

###QQ###

#####

Q#####

#####Q#

#####

##Q#####

#####Q#Q

#Q#####

###QQ###

#####

Q#####

###Q##Q#

#####

##Q#####

#####Q#Q

#Q#####

####Q###

#####

Q#####

###Q##Q#

#####

##Q#####

#####Q##

#Q#####

####Q###

#####Q

Q#####

#####Q#

#####

##Q#####

#####Q##

#Q#####

####Q###

#####Q

Q##Q####

###Q##Q#

#####

##Q#####

#####Q##

#Q#####

####Q###

#####Q

Q#####

#####Q#

#####

##Q#####

#####Q##

#Q#####

####Q###

#####Q

Q##Q####

###Q##Q#

#####

##Q#####

#####Q##

#Q#####

####Q###

#####Q

Q#####

#####Q#

#####

##Q#####

#####Q##

#Q#####

####Q###

#####Q

Q##Q####

#####Q#

Q#####

##Q#####

#####Q##

#Q#####

####Q###

#####Q

###Q####

#####Q#

#####

##Q####

#####Q##

#Q#####

#####Q###

#####Q

Q##Q####

###Q##Q#

#####

##Q####

#####Q##

#Q#####

#####Q###

#####Q

Q#####

###Q####

#####Q#

##Q####

#####Q##

#Q#####

#####Q###

#####Q

Q#####

###Q##Q#

#####

##Q####

#####Q##

#Q#####

####Q###

#####Q

Q#####

#####Q#

#####

##Q#####

#####Q##

#Q#####

####Q###

#####Q

Q##Q####

###Q##Q#

#####

##Q#####

#####Q##

#Q#####

####Q###

#####Q

Q#####

###Q####

#####Q#

##Q#####

#####Q##

#Q#####

####Q###

#####Q

Q#####

###Q##Q#

#####

##Q#####

#####Q##

#Q#####

####Q###

#####Q

Q#####

#####Q#

#####

##Q#####

#####Q##

#Q#####

####Q###

#####Q

Q##Q####

#####Q#

#####

Q#Q#####

#####Q##

#Q#####

####Q###

#####Q

###Q###

#####Q#

##Q#####

Q#####

#####Q##

#Q#####

####Q###

#####Q

###Q####

#####Q#

##Q#####

Q#####

#####Q##

#Q#####

####Q###

#####

###Q###Q

#####Q#

##Q#####

Q#####

#####Q##

#Q#####

####Q###

#####Q

###Q####

#####Q#

##Q#####

Q#####

#####Q##

#Q#####Q

####Q###

#####

###Q####

#####Q#

##Q#####

Q#####

#####Q##

#####Q

####Q###

#Q#####

###Q####

Path cost: 25

Third Path for Sideways Move

###Q#Q##

#####Q#Q#

#####

Q#####

#Q#####

##Q####Q

#####

#####

###Q#Q##

####Q#Q#

#####

Q#####

#####

##Q####Q

#####

#Q#####

###Q#Q##

####Q#Q#

##Q#####

Q#####

#####

#####Q

#####

#Q#####

###Q#Q##

#####Q#

##Q#####

Q#####

#####

#####Q

####Q###

#Q#####

###Q####

#####Q#

##Q#####

Q#####

#####Q##

#####Q

####Q###

#Q#####

###Q####

Q#####Q#

##Q#####

#####

#####Q##

#####Q

####Q###

#Q#####

###Q##Q#

Q#####

##Q#####

#####

#####Q##

#####Q

####Q###

#Q#####

###Q####

Q#####Q#

##Q#####

#####

#####Q##

#####Q

####Q###

#Q#####

###Q####

#####Q#

##Q#####

#####

#####Q##

Q#####Q

####Q###

#Q#####

###Q####

#####Q#

##Q#####

Q#####

#####Q##

#####Q

####Q###

#Q#####

###Q####

#####Q#

##Q#####

#####

#####Q##

Q#####Q

####Q###

#Q#####

###Q####

#####Q#

##Q#####

#####Q

#####Q##

Q#####

####Q###

#Q#####

###Q####

#####Q#

##Q#####

#####Q

#####Q##

Q#####

#####

#Q##Q###

###Q####

#####Q#

##Q#####

#####Q

#####Q##

QQ#####

#####

####Q###

###Q###

#####Q#

##Q#####

#####Q

#####Q##

Q#####

#####

#Q##Q###

###Q###

#####Q#

##Q#####

#####Q

#####Q##

Q#####

####Q###

#Q#####

###Q###

#####Q#

##Q#####

#####

#####Q##

Q#####Q

####Q###

#Q#####

###Q###

#####Q#

##Q#####

Q#####

#####Q##

#####Q

####Q###

#Q#####

###Q####

Q#####Q#

##Q#####

#####

#####Q##

#####Q

####Q###

#Q#####

###Q##Q#

Q#####

##Q#####

#####

#####Q##

#####Q

####Q###

#Q#####

###Q####

Q#####Q#

##Q#####

#####

#####Q##

#####Q

#####Q###

#Q#####

###Q####

#####Q#

##Q#####

#####

#####Q##

Q#####Q

#####Q###

#Q#####

###Q####

#####Q#

##Q#####

#####Q

#####Q##

Q#####

#####Q###

#Q#####

###Q####

#####Q#

##Q#####

#####Q

#####Q##

Q#####

#####

#Q##Q###

###Q####

#####Q#

##Q#####

#####Q

#####Q##

Q#####

####Q###

#Q#####

###Q####

#####Q#

##Q#####

#####

#####Q##

Q#####Q

####Q###

#Q#####

###Q####

#####Q#

##Q#####

Q#####

#####Q##

#####Q

####Q###

#Q#####

###Q####

Q#####Q#

##Q#####

#####

#####Q##

#####Q

####Q###

#Q#####

###Q##Q#

Q#####

##Q#####

#####

#####Q##

#####Q

####Q###

#Q#####

###Q####

Q#####Q#

##Q#####

#####

#####Q##

#####Q

####Q###

#Q#####

###Q##Q#

Q#####

##Q#####

#####

#####Q##

#####Q

####Q###

#Q#####

###Q####

Q#####Q#

##Q#####

#####

#####Q##

#####Q

####Q###

#Q#####

###Q####

#####Q#

##Q#####

Q#####

#####Q##

#####Q

####Q###

#Q#####

###Q####

#####Q#

##Q#####

#####

#####Q##

Q#####Q

#####Q###

#Q#####

###Q####

#####Q#

##Q#####

#####Q

#####Q##

Q#####

#####Q###

#Q#####

###Q####

#####Q#

##Q#Q###

#####Q

#####Q##

Q#####

#####

#Q#####

###Q####

#####Q#

#####Q###

#####Q

#####Q##

Q#####

##Q#####

#Q#####

###Q####

#####Q#

##Q#Q###

#####Q

#####Q##

Q#####

#####

#Q#####

###Q####

#####Q#

##Q#####

#####Q

#####Q##

Q#####

####Q###

#Q#####

###Q####

#####Q#

##Q#####

#####Q

#####Q##

Q#####

#####

#Q##Q###

###Q####

#####Q#

##Q#Q###

#####Q

#####Q##

Q#####

#####

#Q#####

###Q####

#####Q#

##Q#####

#####Q

#####Q##

Q#####

#####

#Q##Q###

###Q####

#####Q#

##Q#####

#####Q

#####Q##

Q#####

####Q###

#Q#####

###Q###

#####Q#

##Q#Q###

#####Q

#####Q##

Q#####

#####

#Q#####

###Q###

#####Q#

#####Q###

#####Q

#####Q##

Q#####

##Q#####

#Q#####

###Q###

#Q####Q#

#####Q###

#####Q

#####Q##

Q#####

##Q#####

#####

###Q###

#Q#####

####Q###

#####Q

#####Q##

Q#####

##Q#####

#####Q#

Path cost: 47

Steepest Ascent : Success Count = 67 Success rate = 0.134 Fail count = 433 Failure rate = 0.866 Avg Success Steps = 5.149253731343284 Avg Fail Steps : 3.9907621247113165

Random Restart Steepest Ascent: Success Count = 500 Success rate = 1.0 Fail Count = 0 Failure rate = 0.0 Avg Success Steps = 23.66 Avg Random Restart = 7.868

Sideways Move : Success Count = 469 Success rate = 0.938 Fail count = 31 Failure rate = 0.062000000000000055 Avg Success Steps = 0.002 Avg Fail Steps = 72.16129032258064

Random Restart Sideways : Success Count = 500 Success rate = 1.0 Fail Count = 0 Failure rate = 0.0 Avg Success Steps = 26.26 Avg Random Restart = 1.054

?

Statistics:

Steepest Ascent :

Success Count = 67

Success rate = 0.134

Fail count = 433

Failure rate = 0.866

Avg Success Steps = 5.149253731343284

Avg Fail Steps : 3.9907621247113165

Random Restart Steepest Ascent:

Success Count = 469

Success rate = 0.938

Fail Count = 0

Failure rate = 0.0

Avg Success Steps = 22.66

Avg Random Restart = 7.868

Sideways Move :

Success Count = 469

Success rate = 0.938

Fail count = 31

Failure rate = 0.062000000000000055

Avg Success Steps = 0.002

Avg Fail Steps = 72.16129032258064

Random Restart Sideways :

Success Count = 500

Success rate = 1.0

Fail Count = 0

Failure rate = 0.0

Avg Success Steps = 26.26

Avg Random Restart = 1.054

Source Code:

Board.py

```
# -*- coding: utf-8 -*-  
"""
```

Created on thursday March 26 2020

@authors: Aditya Kadimi,
 Mrudula Ravipati ,
 Sai Harish Paleti.

```
"""
```

```
import numpy as np
```

```
class Queen:
```

```
    def __init__(self,r,c):
```

```
        self.r=r
```

```
        self.c=c
```

```
    def attack_check(self,q):
```

```
        return self.r ==q.get_rows() or self.c==q.get_columns() or  
abs(self.c - q.get_columns()) == abs(self.r - q.get_rows())
```

```
    def go_down(self,steps):
```

```
        self.r = (self.r + steps) % Board.get_size();
```

```
    def get_rows(self):
```

```
        return self.r
```

```
    def get_columns(self):
```

```
        return self.c
```

```
    def toString(self):
```

```
        return "(" + str(self.r) + ", " + str(self.c) + ")"
```

```
class Board:
```

```
    board_size=8
```

```
    def __init__(self):
```

```

        self.state=[]
        self.next_board=[]
        self.h=0

    def Board(self,n):
        for i in range(Board.board_size):
            self.state.append(Queen(n.state[i].get_rows(),
n.state[i].get_columns()))

    def get_size():
        return Board.board_size

    def set_size(size):
        Board.board_size=size

    def create_board(self, initial_state):
        count=0
        for i in range(Board.board_size):
            for j in range(1,Board.board_size):
                new_board=Board()
                new_board.Board(initial_state)
                self.next_board.insert(count, new_board )
                self.next_board[count].state[i].go_down(j)
                self.next_board[count].calculate_h()
                count+=1

        return self.next_board

    def calculate_h(self):

```

```

    for i in range(Board.board_size-1):
        for j in range(i+1,Board.board_size):
            if (self.state[i].attack_check(self.state[j])):
                self.h+=1
    return self.h

def get_h(self):
    return self.h

def compare(self,n):
    if(self.h<n.get_h()):
        return -1
    elif(self.h>n.get_h()):
        return 1
    else:
        return 0

def set_state_board(self,s):
    for i in range(Board.board_size):
        self.state.append( Queen(s[i].get_rows(),
s[i].get_columns()))

def toString(self):
    result=""
    board = np.zeros((Board.get_size(),Board.get_size()), dtype=str)
    for i in range(Board.board_size):
        for j in range(Board.board_size):
            board[i][j]="#"
    for i in range(Board.board_size):

```

```

board[self.state[i].get_rows()][self.state[i].get_columns()]="Q"
    for i in range(Board.board_size):
        for j in range(Board.board_size):
            result+=board[i][j]
        result += "\n"
    return result

```

n_queens.py

```

# -*- coding: utf-8 -*-

```

```

"""

```

Created on thursday March 26 2020

```

@author: Aditya Kadimi,
        Mrudula Ravipati ,
        Sai Harish Paleti.

```

```

"""

```

```

import numpy as np
import random
from board import Board
from board import Queen
from Steepest_Ascent import Steepest_Ascent
from Sideways_Move import Sideways_Move
from Random_Restart_Steepest_Ascent import
Random_Restart_Steepest_Ascent
from Random_Restart_Sideways import Random_Restart_Sideways

board_size = input("Enter the the value for number of queens
in n-Queen problem: ")

```



```
board_size=int(board_size)
runtime = input("Please enter the Runtime: ")
runtime=int(runtime)
```

```
Board.set_size(board_size)
```

```
def generate_board():
    start=[]
    for i in range(board_size):
        start.append( Queen(random.randint(0,board_size-1)
,i))
    return start
```

```
steepest_ascent_sum_succes=0
steepest_ascent_aver_success=0
steepest_ascent_success_steps=0
steepest_ascent_aver_succes_steps=0
steepest_ascent_faill_steps=0
steepest_ascent_aver_faill_steps=0
```

```
side_move_sum_succes=0
side_move_aver_succes=0
side_move_aver_succes=0
```

side_move_aver_succes_steps=0

side_move_fail_steps=0

side_move_aver_fail_steps=0

random_restart_steepest_ascent_summ_succes=0

random_restart_steepest_ascent_aver_succes=0

random_restart_steepest_ascent_succes_steps=0

random_restart_steepest_ascent_aver_succes_steps=0

random_restart_steepest_ascent_count=0

random_restart_side_move_sum_succes=0

random_restart_side_move_aver_succes=0

random_restart_side_move_aver_succes=0

random_restart_side_move_aver_succes_steps=0

random_restart_side_moves_count=0

for current_test in range(1, runtime+1):

 initial_board= generate_board()

 steepest_ascent= Steepest_Ascent(initial_board)

 random_restart_steepest_ascent =
Random_Restart_Steepest_Ascent(initial_board)

 sideways_move= Sideways_Move(initial_board)

 random_restart_sideways_move=
Random_Restart_Sideways(initial_board)

```

    steepest_ascent_board=
    steepest_ascent.climbing_algorithm()

    random_restart_steepest_ascent_board =
    random_restart_steepest_ascent.climbing_algorithm(initial_board)

    sideways_move_board= sideways_move.climbing_algorithm()

    random_restart_sideways_move_board=
    random_restart_sideways_move.climbing_algorithm(initial_board)

```

```

#steepest Ascent

```

```

    if steepest_ascent_board.calculate_h()==0:
        steepest_ascent_sum_succes+=1
        steepest_ascent_success_steps=
        steepest_ascent.get_steps()

    steepest_ascent_aver_succes_steps+=steepest_ascent_success_steps

```

```

    else:

```

```

    steepest_ascent_faill_steps=steepest_ascent.get_steps()

    steepest_ascent_aver_faill_steps +=
    steepest_ascent_faill_steps

```

```

    if current_test==33:
        print("First Path for Steepest Ascent")
        x = steepest_ascent.list_to_print()

```

```
steepest_ascent.print_path(x)
print("Path cost: ", len(x))
```

```
if current_test==97:
    print("Second Path for Steepest Ascent")
    x = steepest_ascent.list_to_print()
    steepest_ascent.print_path(x)
    print("Path cost: ", len(x))
```

```
if current_test==139:
    print("Third Path for Steepest Ascent")
    x = steepest_ascent.list_to_print()
    steepest_ascent.print_path(x)
    print("Path cost: ",len(x))
```

```
#Random Restart Steepest Ascent
```

```
if random_restart_steepest_ascent_board.get_h() == 0 :
    random_restart_steepest_ascent_summ_succes+=1
```

```
random_restart_steepest_ascent_succes_steps=random_restart_steepest_ascent.get_step_count()
```

```
random_restart_steepest_ascent_aver_succes_steps+=random_restart_steepest_ascent_succes_steps
```

```
random_restart_steepest_ascent_count+=random_restart_steepest_ascent.get_random_used()
```

```

#Sideways move
if sideways_move_board.get_h() == 0:
    side_move_sum_succes+=1
    side_move_aver_succes=sideways_move.get_step_count()
    side_move_aver_succes_steps+=side_move_aver_succes

else:
    side_move_fail_steps=sideways_move.get_step_count()
    side_move_aver_fail_steps+=side_move_fail_steps

if current_test==181:
    print("First Path for Steepest Ascent Sideways Move")
    x = sideways_move.list_to_print()
    sideways_move.print_path(x)
    print("Path cost: ",len(x))
if current_test==214:
    print("Second Path for Sideways Move")
    x = sideways_move.list_to_print()
    sideways_move.print_path(x)
    print("Path cost: ",len(x))

if current_test==376:
    print("Third Path for Sideways Move")
    x = sideways_move.list_to_print()
    sideways_move.print_path(x)
    print("Path cost: ",len(x))

```

```

#Random Restart without sideways move
if random_restart_sideways_move_board.get_h() == 0:
    random_restart_side_move_sum_succes+=1

random_restart_side_move_aver_succes=random_restart_sideways_m
ove.get_step_count()

    random_restart_side_move_aver_succes_steps+=
random_restart_side_move_aver_succes;

random_restart_side_moves_count+=(random_restart_sideways_move
.get_random_used());

steepest_ascent_aver_success=steepest_ascent_sum_succes/runtim
e

random_restart_steepest_ascent_aver_succes =
random_restart_steepest_ascent_summ_succes / runtime;
side_move_aver_succes= side_move_sum_succes/ runtime
random_restart_side_move_aver_succes
=random_restart_side_move_sum_succes / runtime;

print("Steepest Ascent :")

        + " Success Count = " ,
steepest_ascent_sum_succes
        , " Success rate = " ,
steepest_ascent_aver_success
        , " Fail count = " , (runtime -

```

```

steepest_ascent_sum_succes)
        , " Failure rate = " , (1 -
steepest_ascent_aver_success)
        , " Avg Success Steps = " ,
(steepest_ascent_aver_succes_steps/steepest_ascent_sum_succes)
        , " Avg Fail Steps : " ,
((steepest_ascent_aver_fail_steps)/(runtime-
steepest_ascent_sum_succes)));

```

```

print("Random Restart Steepest Ascent:"

```

```

        , " Success Count = " ,
random_restart_steepest_ascent_summ_succes
        , " Success rate = " ,
random_restart_steepest_ascent_aver_succes
        , " Fail Count = " , (runtime -
random_restart_steepest_ascent_summ_succes)
        , " Failure rate = " , (1 -
random_restart_steepest_ascent_aver_succes)
        , " Avg Success Steps = " ,
((random_restart_steepest_ascent_aver_succes_steps)/runtime)
        , " Avg Random Restart =" ,
(random_restart_steepest_ascent_count/runtime));

```

```

print("Sideways Move : "

```

```

        , " Success Count = " ,
side_move_sum_succes
        , " Success rate = " ,
side_move_aver_succes

```

```

        , " Fail count = " , (runtime -
side_move_sum_succes)
        , " Failure rate = " , (1 -
side_move_aver_succes)
        , " Avg Success Steps = " ,
(side_move_aver_succes/side_move_sum_succes)
        , " Avg Fail Steps = " ,
(np.float64(side_move_aver_fail_steps)/(runtime-
side_move_sum_succes)));

```

```

print("Random Restart Sideways :")

```

```

        , " Success Count = " ,
random_restart_side_move_sum_succes
        , " Success rate = " ,
random_restart_side_move_aver_succes
        , " Fail Count = " , (runtime -
random_restart_side_move_sum_succes)
        , " Failure rate = " , (1 -
random_restart_side_move_aver_succes)
        , " Avg Success Steps = " ,
((random_restart_side_move_aver_succes_steps)/runtime)
        , " Avg Random Restart = " ,
(random_restart_side_moves_count)/runtime);

```


Random_Restart_Sideways.py

```
# -*- coding: utf-8 -*-
```

```
"""
```

Created on thursday March 26 2020

@authors: Aditya Kadimi, Mrudula Ravipati , Sai Harish Paleti.

```
"""
```

```
import random
```

```
from board import Board
```

```
from board import Queen
```

```
from Sideways_Move import Sideways_Move
```

```
class Random_Restart_Sideways:
```

```
    def __init__(self,s):
```

```
        self.step=0
```

```
        self.begin=0
```

```
        self.sideway_move_object= Sideways_Move(s)
```

```
        Random_Restart_Sideways.restart_used=1
```

```
    def climbing_algorithm(self,s):
```

```
        curr_board=self.sideway_move_object.get_start_board()
```

```
        self.set_start_board(curr_board)
```

```
        h= curr_board.get_h()
```

```
        self.step=0
```

```

        while h!=0:
            next_board=
self.sideway_move_object.climbing_algorithm()
            self.step+=
self.sideway_move_object.get_step_count()
            h = next_board.get_h()

        if h!=0:
            s=Random_Restart_Sideways.generate_board()
            self.sideway_move_object= Sideways_Move(s)
            Random_Restart_Sideways.restart_used+=1
        else:
            curr_board=next_board
    return curr_board

```

```

def generate_board():
    start=[]
    for i in range(8):
        start.append(
Queen(random.randint(0,Board.get_size()-1) ,i))
    return start

```

```

def set_start_board(self, curr_board):
    self.begin = curr_board
def get_step_count(self):
    return self.step
def get_random_used(self):
    return Random_Restart_Sideways.restart_used

```

Random_Restart_Steepest_Ascent.py

```
# -*- coding: utf-8 -*-
```

```
"""
```

```
Created on thursday March 26 2020
```

```
@authors: Aditya Kadimi,  
          Mrudula Ravipati ,  
          Sai Harish Paleti.
```

```
"""
```

```
import random
```

```
from board import Board
```

```
from board import Queen
```

```
from Steepest_Ascent import Steepest_Ascent
```

```
class Random_Restart_Steepest_Ascent:
```

```
    def __init__(self,s):
```

```
        self.step=0
```

```
        self.start=0
```

```
        self.steepest_ascent_obj= Steepest_Ascent(s)
```

```
        Random_Restart_Steepest_Ascent.restart_used=1
```

```
    def climbing_algorithm(self,s):
```

```
        curr_board=self.steepest_ascent_obj.get_start_board()
```

```

self.set_start_board(curr_board)
h= curr_board.get_h()
self.step=0

while h!=0:
    next_board=
self.steepest_ascent_obj.climbing_algorithm()
    self.step+= self.steepest_ascent_obj.get_steps()
    h = next_board.get_h()

    if h!=0:

s=Random_Restart_Steepest_Ascent.generate_board()
    self.steepest_ascent_obj= Steepest_Ascent(s)
    Random_Restart_Steepest_Ascent.restart_used+=1
else:
    curr_board=next_board
    self.step-=
self.steepest_ascent_obj.get_steps()
    Random_Restart_Steepest_Ascent.restart_used+=1
return curr_board

```

```

def generate_board():
    start=[]
    for i in range(8):
        start.append(
Queen(random.randint(0,Board.get_size()-1) ,i))

```

```

        return start
    def set_start_board(self, curr_board):
        self.start = curr_board
    def get_step_count(self):
        return self.step
    def get_random_used(self):
        return Random_Restart_Steepest_Ascent.restart_used

```

Sideways_Move.py

```

# -*- coding: utf-8 -*-

```

```

"""

```

Created on thursday March 26 2020

```

@authors: Aditya Kadimi,
          Mrudula Ravipati ,
          Sai Harish Paleti.

```

```

"""

```

```

import random
from board import Board
from board import Queen

```

```

class Sideways_Move:

```

```

    def __init__(self,s):
        first_state=[]
        self.initial=Board()
        self.step=0

```

```

        self.print_node=[]
        for i in range(Board.get_size()):

first_state.append((Queen(s[i].get_rows(),s[i].get_columns()))
)
        self.initial.set_state_board(first_state)
        self.initial.calculate_h()

def climbing_algorithm(self):
    current_board=self.initial
    count=0

    while True:

successors=current_board.create_board(current_board)
        select_random_successors=[]

        exist_better =False;
        exist_best=False

        self.print_node.append(current_board)

        for i in range(len(successors)):
            if count==100:
                break
            if(successors[i].compare(current_board) <= 0):
                if(successors[i].compare(current_board) <
0):

```

```

        count=0
        select_random_successors=[]
        current_board=successors[i]
        exist_better=True
        self.step+=1
    elif(successors[i].compare(current_board) == 0):

select_random_successors.append(successors[i])

        if not exist_better and not not
select_random_successors:

            current_board=
select_random_successors[random.randint(0,len(select_random_su
ccessors))-1]
            exist_best=True
            count +=1
            self.step+=1
        if not exist_best and not exist_better:

            return current_board

def get_start_board(self):
    return self.initial

def print_path(self,print_nodes):
    for i in range(len(self.print_node)):
        print(self.print_node[i].toString())

```

```
def list_to_print(self):  
    return self.print_node
```

```
def get_step_count(self):  
    return self.step
```

Steepest_Ascent.py

```
# -*- coding: utf-8 -*-
```

```
"""
```

Created on thursday March 26 2020

```
@authors: Aditya Kadimi,  
          Mrudula Ravipati ,  
          Sai Harish Paleti.
```

```
"""
```

```
import random
```

```
from board import Board
```

```
from board import Queen
```

```
class Steepest_Ascent:
```



```

def __init__(self,s):
    self.step=0
    self.print_nodes=[]
    self.start_board= Board()
    start_state= []
    for i in range(Board.get_size()):

start_state.append((Queen(s[i].get_rows(),s[i].get_columns())))
)

    self.start_board.set_state_board(start_state)
    self.start_board.calculate_h()


def climbing_algorithm(self):
    curr_board=self.start_board

    while True:
        successors=curr_board.create_board(curr_board)
        exist_better = False
        self.print_nodes.append(curr_board)
        self.step+=1

        for i in range(len(successors)):
            if(successors[i].compare(curr_board) < 0):

```

```
        curr_board=successors[i]
        exist_better=True

        if not exist_better:
            return curr_board
def list_to_print(self):
    return self.print_nodes

def print_path(self,print_nodes):
    for i in range(len(self.print_nodes)):
        print(self.print_nodes[i].toString())

def get_start_board(self):
    return self.start_board

def get_steps(self):
    return self.step
```

Number of Queens	Search Used	Success Rate and Number of steps	Failure Rate and Number of steps	Number of Restarts
8	Hill-Climbing Steepest Accent	Rate: 0.134 Steps: 5.149	Rate: 0.866 Steps: 3.99	No Restarts
8	Random-restart without Sideway moves	Rate: 1.00 Steps: 23.66	Rate: 0.00 Steps: 0.00	7.868
8	Hill-Climbing with Sideway moves	Rate: 0.938 Steps: 0.002	Rate: 0.062 Steps: 72.161	No Restarts
8	Random-restart with Sideway moves	Rate: 1.00 Steps: 26.26	Rate: 0.00 Steps: 0.00	1.054

OBSERVATIONS

The success rate is highest when Hill Climbing with sideways method is used and it reduces drastically from 93.8% to 13.4% when steepest ascent method is used. The failure rate reduces from 86.60% to 0.062% when Hill Climbing with sideways method is used.