|  |  |  |
| --- | --- | --- |
|  | 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](mailto:akadimi@uncc.edu)

801149978

Mrudula Ravipati

mravipat@uncc.edu

801167901

Sai Harish Paleti

[spaleti@uncc.edu](mailto: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](https://en.wikipedia.org/wiki/Iterative_algorithm) that starts with an arbitrary solution to a problem, then attempts to find a better solution by making an [incremental](https://en.wikipedia.org/wiki/Incremental_heuristic_search) 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 thenode 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**

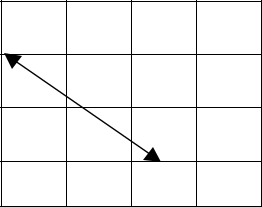
The number of misplaced tiles calculated by comparing the current state and goal state.

**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.



**Q**

**Q**

**Q**

**Q**

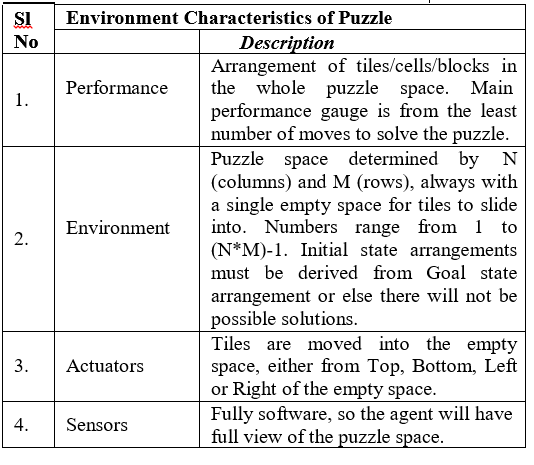
The heuristic value for the above problem is four since there are four pairs of queens that are attacking each other at this moment.

**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.



**Hill Climbing Algorithm**

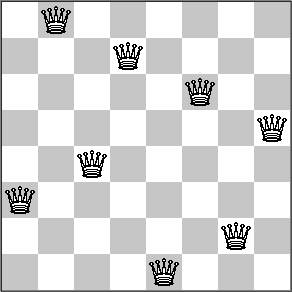
Hill Climbing works disregards memory of explored nodes. Therefore, it travels down the Search Tree by selecting the successor with the cheapest heuristics value, without retaining memory of explored states. This will ensure that the heuristics technique functions with minimal use of memory, least computation 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×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\*n chess board as a matrix.

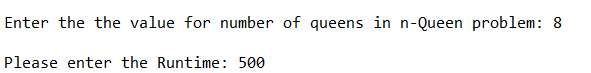
The classic combinatorial problem is to place N-Queens on a chessboard so that no two attack each other. In the chess Queens attacking in three directions i.e. horizontally, vertically and diagonally. The problem can be generalized as placing ‘n’ non attacking queens on an N x N chessboard. Since each 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 (q1, q2, …, qn) that are permutations of an n-tuple (1, 2, 3, …, 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**

**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

@authors: 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\_faiil\_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\_faiil\_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\_move.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/runtime

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\_faiil\_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\_successors))-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 |

**OBESERVATIONS**

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.