**REPORT**

**Project 2: Hill-climbing Algorithm and Its Variants**

**(N-queens problem)**

**Submitted By**

Jawad Chowdhury, (ID# 801135477)

**N-Queen Problem Formulation**

Here, in this project, we have worked on the implementation of an 8-Queen Problem which is a specific version of n-queen problem solving. To solve this problem, we have used Hill-Climbing search algorithm and its different variants.

In a general sense, the problem gives us an initial state (a random formation of the 8-queens in an 8X8 board) and the constraint is that none of these queens can attack each other.

To solve this problem, we can use a heuristic where the cost is based on the number of collisions between all the queens of the current state and the Hill Climbing algorithms strategy is to explore the neighboring states in such an ascent method so that reduces the cost associated with the heuristic.

There are different strategies to take when we get stuck in a state where all possible neighbors give us higher costs. Based on the different strategies, we have implemented the 1. Basic 2. Sideways 3. Random Restart (with/without sideways) variants of the Hill Climbing Algorithm.

**Program Structure (Variables, Functions & Procedures)**

In my implementation, I have used 4 separate scripts to isolate the implementation of Basic, Sideways, Random Restart (with sideways), Random Restart (without sideways). Mostly the generic structure of these scripts is that they contain 2 basic classes. One of them is to maintain the basic flow of the problem solving which is being denoted as ‘**NQueen**’ and the other one is the ‘**Board**’ class to represent different states of the board.

**Board:** The class Board is comparatively the simpler one. It consists based on the **‘str()’** method to represent each state of the Board and ‘**get\_hcost()**’ method which is being used to calculate the **heuristic cost** of that specific state. Now this method is a combination of 2 other methods to calculate the diagonal and horizontal/vertical number of collisions in a state.

**NQueen:** This class contains method to apply the hill climbing algorithm (**hill\_climbing()**). Another important method of this class (**get\_best\_neighbor()**) is the method to get the best successor according to the strategy based on different variants of the algorithm.

The **NQueen** class contains a **run()** function which is the basic procedure to run the Hill Climbing algorithm based on the value of ‘**n**’ and **number of runs** and these values are being taken from the file named ‘**input.txt**’. The first value in the file represents the value of ‘**n**’ and in the following line, the value contains the number of run as ‘**no\_runs**’.

**Execution Results (According to the Rubric)**

**Source Code**

I have 4 separated scripts for different variants (1 for basic, 1 for sideways, 1 for random restart with sideways, 1 for random restart without sideways) just to simplify rather than having all the variants in one script. These scripts are mostly similar. The major difference is on the section of **hill\_climbing()** method based on the output from the ‘**get\_best\_neightbor()**’ method. So, in the following I have given the source code of Hill Climbing algorithm’s basic version.

===========================================================

*"""  
 Implementation of hill climbing search and its variants.  
 author: Jawad Chowdhury.  
"""***import** random, copy  
  
**class** Board:  
 *"""  
 This class maintains properties related to the different state of the N-Queen Problem.  
 """* **def** \_\_init\_\_(self, n, state=**None**):  
 self.n = n  
 **if** state==**None**:  
 self.state = [[**'\_' for** c **in** range(n)] **for** r **in** range(n)]  
 **for** q **in** range(n):  
 r, c = random.randint(0,n-1), random.randint(0,n-1)  
 **while** self.state[r][c] != **'\_'**:  
 r, c = random.randint(0, n - 1), random.randint(0, n - 1)  
 self.state[r][c] = **'Q'  
 else**:  
 self.state=state  
 self.hcost = self.get\_hcost()  
 **def** get\_hvcost(self):  
 hvcost = 0  
 **for** r **in** range(self.n):  
 **for** c **in** range(self.n):  
 **if** self.state[r][c] == **'Q'**:  
 **for** t **in** range(self.n):  
 **if** self.state[r][t] == **"Q"**:  
 hvcost += 1  
 **if** self.state[t][c] == **"Q"**:  
 hvcost += 1  
 hvcost = hvcost -2  
 hvcost = hvcost/2  
 **return** hvcost  
 **def** get\_dcost(self):  
 dcost = 0  
 **for** r **in** range(self.n):  
 **for** c **in** range(self.n):  
 **if** self.state[r][c] == **'Q'**:  
 tr, tc = r - 1, c - 1  
 **while** tr >= 0 **and** tc >= 0:  
 **if** self.state[tr][tc] == **"Q"**:  
 dcost += 1  
 tr -= 1  
 tc -= 1  
 tr, tc = r + 1, c - 1  
 **while** tr < self.n **and** tc >= 0:  
 **if** self.state[tr][tc] == **"Q"**:  
 dcost += 1  
 tr += 1  
 tc -= 1  
 tr, tc = r + 1, c + 1  
 **while** tr < self.n **and** tc < self.n:  
 **if** self.state[tr][tc] == **"Q"**:  
 dcost += 1  
 tr += 1  
 tc += 1  
 tr, tc = r - 1, c + 1  
 **while** tr >= 0 **and** tc < self.n:  
 **if** self.state[tr][tc] == **"Q"**:  
 dcost += 1  
 tr -= 1  
 tc += 1  
 dcost = dcost/2  
 **return** dcost  
 **def** get\_hcost(self):  
 *"""  
 This method is being used to calculate the heuristic cost of the current board.* **:return***:  
 """* hvcost = self.get\_hvcost()  
 dcost = self.get\_dcost()  
 hcost = hvcost + dcost  
 **return** hcost  
 **def** \_\_str\_\_(self):  
 *"""  
 String representation of the state.* **:return***:  
 """* s=**''  
 for** r **in** range(self.n):  
 **for** c **in** range(self.n):  
 s += str(self.state[r][c]) + **' '** s += **'\n'** s += str(**'hcost'**) + **' : '** + str(self.hcost)  
 s += **'\n'  
 return** s  
  
**class** NQueen:  
 *"""  
 This class is being used to maintain the overall flow of the N-Queen problem.  
 """* **def** \_\_init\_\_(self, no\_runs, n, variant=**'basic'**):  
 self.no\_runs = no\_runs  
 self.n = n  
 self.variant = variant  
 self.no\_success = 0  
 self.no\_total\_steps = 0  
 self.no\_success\_steps = 0  
 **def** run(self):  
 **for** i **in** range(0, self.no\_runs):  
 print()  
 print(**'========== BOARD :%s =========='**%(i,) )  
 b = Board(n=n)  
 self.hill\_climbing(variant=self.variant, board=b)  
 **def** get\_best\_neighbor(self, board):  
 *"""  
 This method gives the best successor based on the strategy.* **:param** *board:* **:return***:  
 """* best\_board = board  
 best\_cost = board.hcost  
 **for** r **in** range(0,self.n):  
 **for** c **in** range(0,self.n):  
 **if** board.state[r][c] == **'Q'**:  
 **for** nr **in** range(0,self.n):  
 **for** nc **in** range(0,self.n):  
 **if** board.state[nr][nc] == **'\_'**:  
 neighbor\_state = copy.deepcopy(board.state)  
 neighbor\_state[r][c] = **'\_'** neighbor\_state[nr][nc] = **'Q'** neighbor = Board(n=self.n, state=neighbor\_state)  
 **if** neighbor.hcost < best\_cost:  
 best\_cost = neighbor.hcost  
 best\_board = neighbor  
 **return** best\_board, best\_cost  
 **def** hill\_climbing(self, variant=**'basic'**, board=**None**):  
 *"""  
 This method runs the hill climbing algorithm based on the variant.* **:param** *variant:* **:param** *board:* **:return***:  
 """* **if** board **and** variant == **'basic'**:  
 current\_board = board  
 no\_local\_steps = 0  
 **while True**:  
 print(current\_board)  
 best\_neighbor, \_ = self.get\_best\_neighbor(current\_board)  
 **if** current\_board.hcost == best\_neighbor.hcost:  
 **break** no\_local\_steps += 1  
 current\_board = best\_neighbor  
 print(best\_neighbor)  
 **if** best\_neighbor.hcost != 0:  
 print(**'SOLUTION NOT FOUND!!!'**)  
 self.no\_total\_steps += no\_local\_steps  
 **else**:  
 print (**'SOLUTION FOUND!!!'**)  
 self.no\_success += 1  
 self.no\_success\_steps += no\_local\_steps  
 self.no\_total\_steps += no\_local\_steps  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 print(**'Hill Climbing Search (basic)!!!'**)  
 input\_file\_name = **'input.txt'  
 with** open(input\_file\_name) **as** f:  
 lines = f.readlines()  
 values = [line.replace(**'\n'**, **''**).replace(**' '**, **''**) **for** line **in** lines]  
 values = [int(v) **for** v **in** values]  
 n = values[0] *# value of n* no\_run = values[1] *# value of number of runs.* nq\_basic = NQueen(no\_runs=no\_run, n=n, variant=**'basic'**)  
 nq\_basic.run()  
 print()  
 nr = nq\_basic.no\_runs  
 ns = nq\_basic.no\_success  
 rs = (ns/nr)\*100  
 nf = nr-ns  
 rf = (nf/nr)\*100  
 n\_total\_steps = nq\_basic.no\_total\_steps  
 n\_success\_steps = nq\_basic.no\_success\_steps  
 n\_failure\_steps = n\_total\_steps - n\_success\_steps  
 avg\_steps\_success = n\_success\_steps/ns **if** ns != 0 **else** 0  
 avg\_steps\_failure = n\_failure\_steps/nf **if** nf != 0 **else** 0  
 print(**'No of Total Runs: {:.2f}'**.format(nr) )  
 print(**'Success Rate: {:.2f} %'**.format(rs) )  
 print(**'Failure Rate: {:.2f} %'**.format(rf) )  
 print(**'Avg steps at Success: {:.2f} '**.format(avg\_steps\_success) )  
 print(**'Avg steps at Failure: {:.2f} '**.format(avg\_steps\_failure) )

=====================================================================================