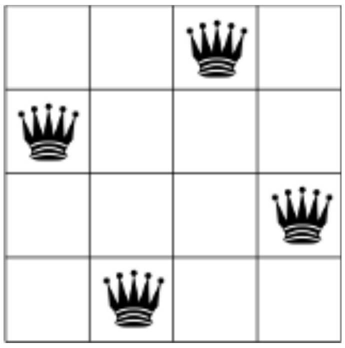
Problem Description: N-Queens Problem

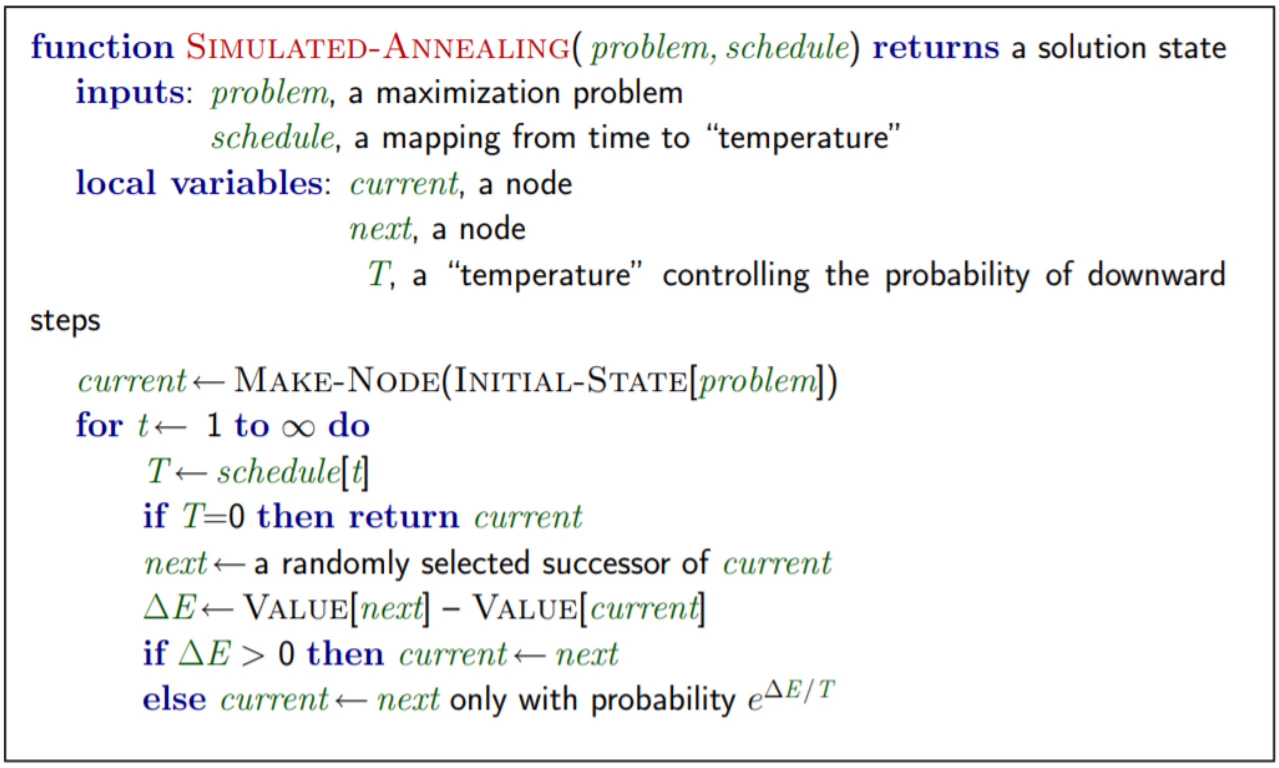


The N-Queens problem originates to a question in chess: can we find a possible solution to place N queens in an NxN chessboard where N>3, so that no two queens can attack one another. To be more clear, a queen which is a piece in chess can move any distance vertically, horizontally or diagonally. Eventually, desired aim is to find non-conflicting states of queens.

Simulated Annealing Description

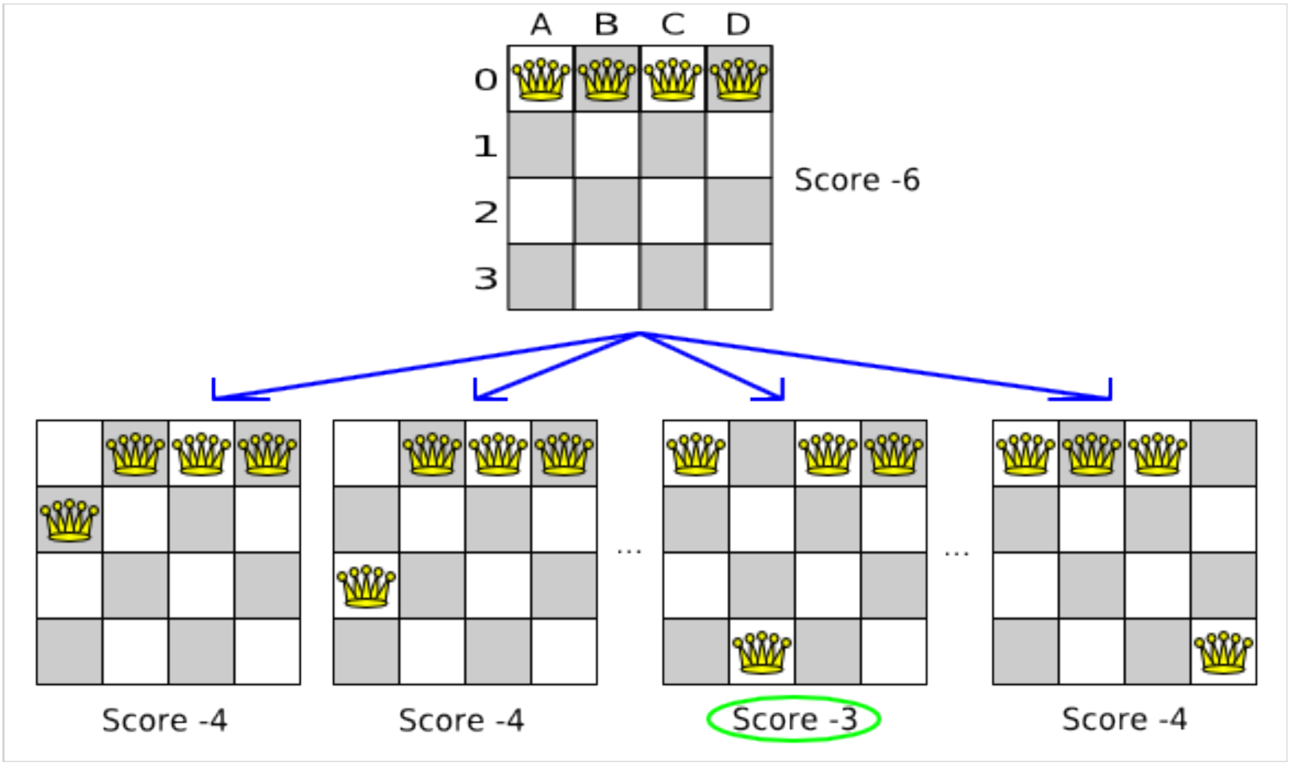
There are different approaches to solve the N-Queens problem such as Hill Climbing, Simulated Annealing, Local Beam Search, Genetic Algorithm. Our choice as a meta-heuristic algorithm was Simulated Annealing which is a popular Monte Carlo algorithm. This algorithm follows a “annealing" process simulates a heated metal. The metal is cooled with a factor gradually and reaches to a minimal energy state also known as very strong state.

In the manner of programming, this technique allows the queens to make random movements at high temperature. On the other hand, occurrence of random moves are limited at low temperature. To be more precise, the move indicates change of the current state of the board to a next state, in which conflict between queens either increase resulting into a worse configuration of the board or may decrease resulting into a better configuration of the board. In terms of avoiding local minimum points, in which neighbor configurations of the board do not yield less conflicting states, at high temperatures Simulated Annealing accepts worse configurations of the board such that global minimum can be reached. As temperature decreases by a chosen cooling factor, selection of worse configurations as next states have less chance to be accepted. Algorithm termination depends on both temperature and total conflict amount of the queens.



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a maximization problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a "temperature" controlling the probability of downward
steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T=0 then return current
        next ← a randomly selected successor of current
        ΔE ← VALUE[next] – VALUE[current]
        if ΔE > 0 then current ← next
        else current ← next only with probability $e^{\Delta E/T}$
```

In our algorithm, queens are placed on chessboard semi-randomly, in which columns are chosen sequentially and rows are chosen randomly. Such a placement provides algorithm to focus only on decreasing row and diagonal conflicts, as column wise conflicts are avoided. Furthermore, next state selection is done in a way that a random queen is selected and moved to a random row without changing its column. Previous conflict result and new conflict result are compared instead of calculating total cost every iteration of the algorithm, and algorithm decides whether to become new state or remain in its previous state. If next state has less conflicts, algorithm proceeds with the new state, otherwise algorithm proceeds with the next state with a certain probability depending on change in temperature and the total conflict amount of the queens. Algorithm continues until either temperature or the total conflict amount becomes zero.



Simulated Annealing Analysis

The performance of Simulated Annealing regardless of N-Queen problem depends on its initial temperature and cooling factor.
As we randomly initialize our board, there is a chance for the board configuration to be already in global minimum. In that case, the conflict amount is already zero such that Simulated Annealing will be terminated directly. Hence we have to calculate conflict amount for each queen at least once, the best case running time of Simulated Annealing will be O(N^2).

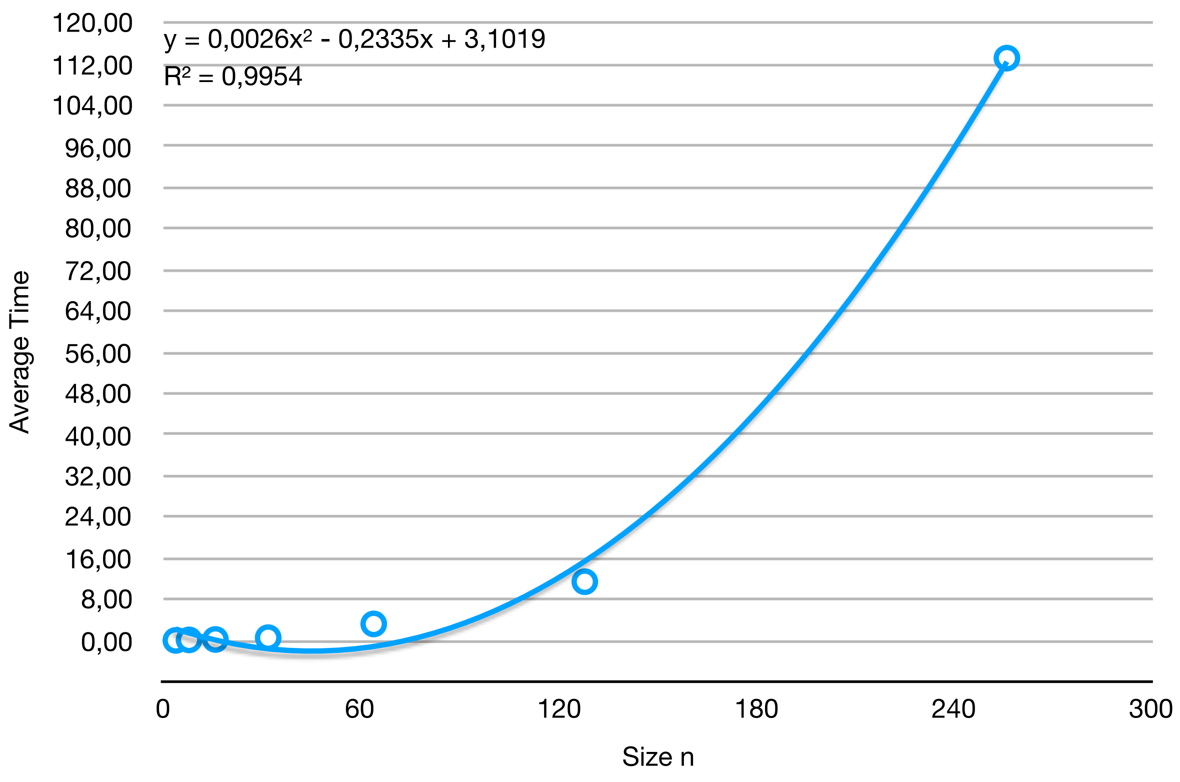
As a consequence of our exponential multiplicative cooling function, worst case running time of the algorithm will be $O(N^2 + N * log_c(\epsilon/T))$ where c is the cooling factor(0<c<1), T is the initial temperature and $\epsilon \approx 0$. Since in the worst case $\epsilon = T * c^m$ where m refers to number of iterations within Simulated Annealing.

Experimental Analysis

3 RUN PER INPUT SIZE

Size	Time	StandartDeviation	StandartError	%90-CL	%95-CL
4	0.00	0.00	0.00	(0.0, 0.0)	(0.0, 0.0)
8	0.10	0.02	0.01	(0.08, 0.11)	(0.08, 0.12)
16	0.18	0.02	0.01	(0.15, 0.20)	(0.15, 0.21)
32	0.53	0.15	0.09	(0.39, 0.68)	(0.37, 0.70)
64	3.20	1.77	1.02	(1.52, 4.88)	(1.20, 5.20)
128	11.40	1.55	0.89	(9.93, 12.87)	(9.65, 13.15)
256	113.13	31.88	18.41	(82.85, 143.41)	(77.05, 149.21)

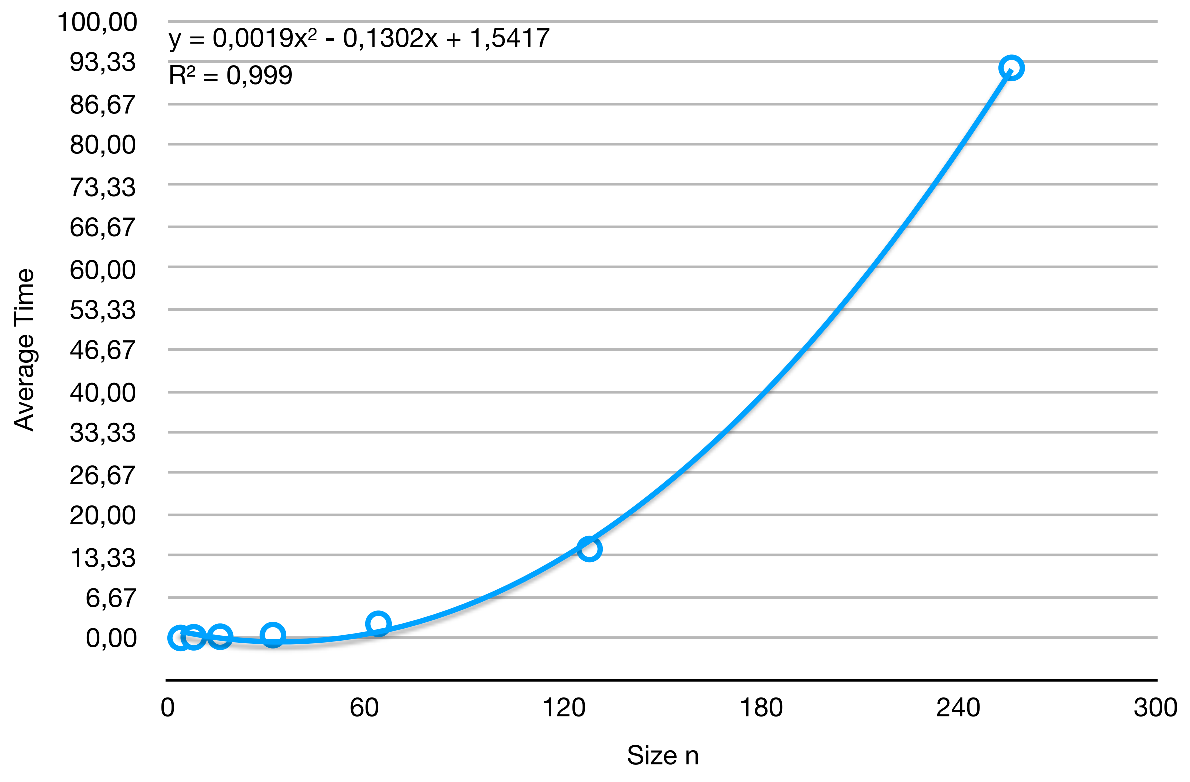
3 RUN PER INPUT SIZE



100 RUN PER INPUT SIZE

Size	Time	StandartDeviation	StandartError	%90-CL	%95-CL
4	0.00	0.00	0.00	(0.0, 0.0)	(0.0, 0.0)
8	0.11	0.02	0.00	(0.11, 0.11)	(0.11, 0.11)
16	0.20	0.04	0.00	(0.19, 0.21)	(0.19, 0.21)
32	0.46	0.16	0.02	(0.44, 0.49)	(0.43, 0.49)
64	2.29	1.55	0.15	(2.04, 2.55)	(1.99, 2.60)
128	14.41	9.76	0.98	(12.80, 16.01)	(12.49, 16.32)
256	92.36	35.07	3.51	(86.59, 98.13)	(85.48, 99.23)

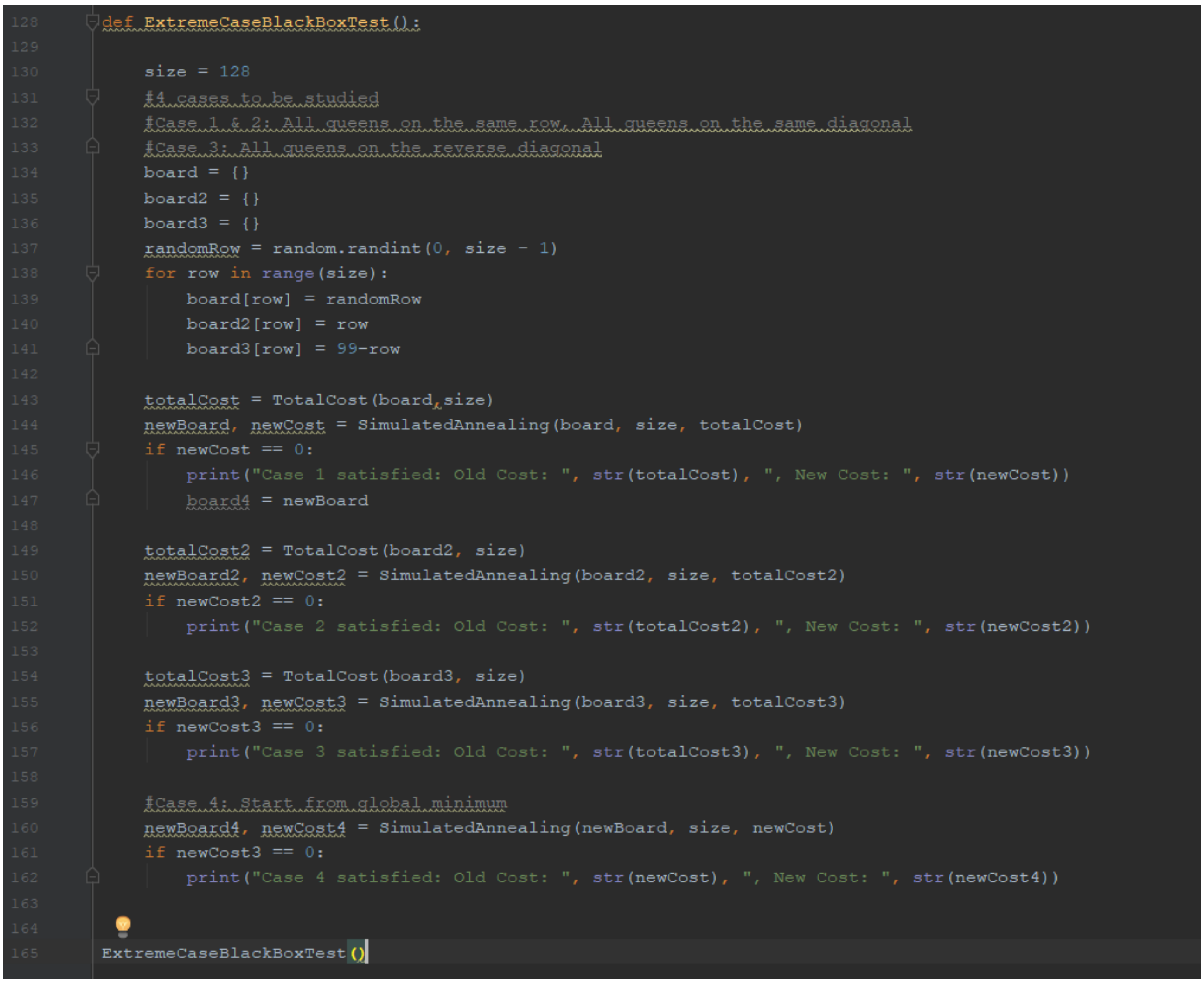
100 RUN PER INPUT SIZE



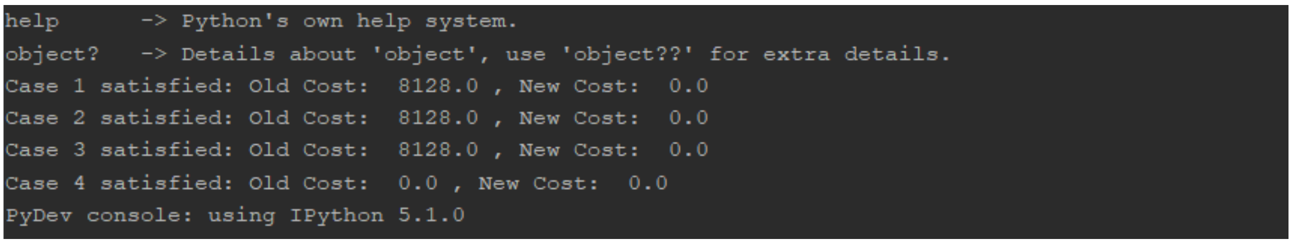
Testing

Our intuition was using Black Box as test method. For 4-Queen Problem, all the permutations of queen placements were taken and tested. All the permutations were able to succeed and decrease their conflict amount up to zero, indicating success for Black Box test.

(a) For extreme cases:



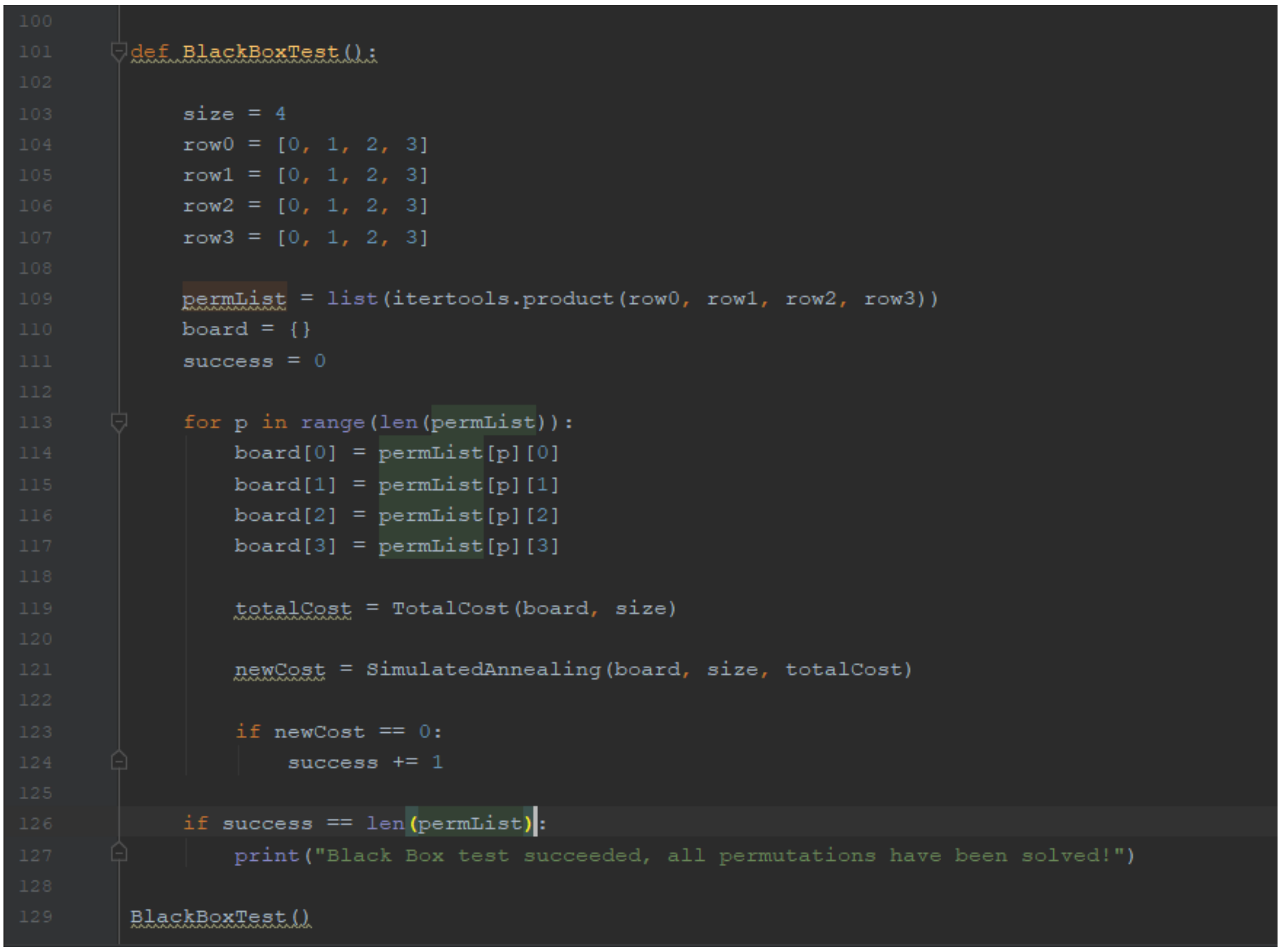
```python
def ExtremeCaseBlackBoxTest():

    size = 128
    #4 cases to be studied
    #Case 1 & 2: All queens on the same row, All queens on the same diagonal
    #Case 3: All queens on the reverse diagonal
    board = {}
    board2 = {}
    board3 = {}
    randomRow = random.randint(0, size - 1)
    for row in range(size):
        board[row] = randomRow
        board2[row] = row
        board3[row] = 99-row

    totalCost = TotalCost(board,size)
    newBoard, newCost = SimulatedAnnealing(board, size, totalCost)
    if newCost == 0:
        print("Case 1 satisfied: Old Cost: ", str(totalCost), ", New Cost: ", str(newCost))
        board4 = newBoard

    totalCost2 = TotalCost(board2, size)
    newBoard2, newCost2 = SimulatedAnnealing(board2, size, totalCost2)
    if newCost2 == 0:
        print("Case 2 satisfied: Old Cost: ", str(totalCost2), ", New Cost: ", str(newCost2))

    totalCost3 = TotalCost(board3, size)
    newBoard3, newCost3 = SimulatedAnnealing(board3, size, totalCost3)
    if newCost3 == 0:
        print("Case 3 satisfied: Old Cost: ", str(totalCost3), ", New Cost: ", str(newCost3))

    #Case 4: Start from global minimum
    newBoard4, newCost4 = SimulatedAnnealing(newBoard, size, newCost)
    if newCost3 == 0:
        print("Case 4 satisfied: Old Cost: ", str(newCost), ", New Cost: ", str(newCost4))


ExtremeCaseBlackBoxTest()
```



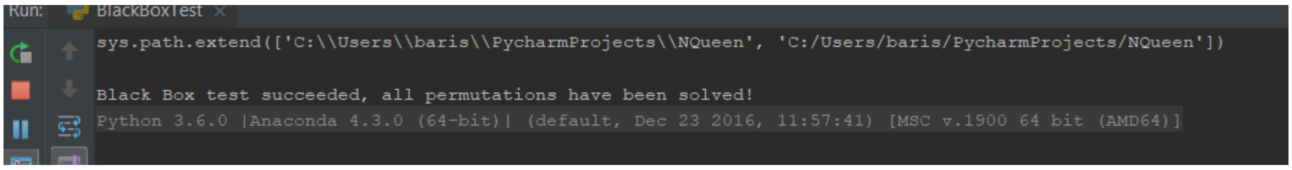
```
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
Case 1 satisfied: Old Cost:  8128.0 , New Cost:  0.0
Case 2 satisfied: Old Cost:  8128.0 , New Cost:  0.0
Case 3 satisfied: Old Cost:  8128.0 , New Cost:  0.0
Case 4 satisfied: Old Cost:  0.0 , New Cost:  0.0
PyDev console: using IPython 5.1.0
```

Extreme case 1: All queens on the same row
Extreme case 2 & 3: All queens on the same diagonal
Extreme case 4: Optimal board configuration

(b) For all permutations:



```python
def BlackBoxTest():

    size = 4
    row0 = [0, 1, 2, 3]
    row1 = [0, 1, 2, 3]
    row2 = [0, 1, 2, 3]
    row3 = [0, 1, 2, 3]

    permList = list(itertools.product(row0, row1, row2, row3))
    board = {}
    success = 0

    for p in range(len(permList)):
        board[0] = permList[p][0]
        board[1] = permList[p][1]
        board[2] = permList[p][2]
        board[3] = permList[p][3]

        totalCost = TotalCost(board, size)

        newCost = SimulatedAnnealing(board, size, totalCost)

        if newCost == 0:
            success += 1

    if success == len(permList):
        print("Black Box test succeeded, all permutations have been solved!")

BlackBoxTest()
```



```
sys.path.extend(['C:\\Users\\baris\\PycharmProjects\\NQueen', 'C:/Users/baris/PycharmProjects/NQueen'])

Black Box test succeeded, all permutations have been solved!
Python 3.6.0 |Anaconda 4.3.0 (64-bit)| (default, Dec 23 2016, 11:57:41) [MSC v.1900 64 bit (AMD64)]
```

As observed, the algorithm finds solutions successfully for all permutation of queens states on 4x4 chessboard.

Conclusions

Through the implementation of algorithm, we used various parameter values for cooling factor and initial temperature which are two significant factors for the heuristic. Our optimal value for initial temperature was 18.000 and optimal value for cooling factor was 0.999. We run the algorithm for the matrix size range from 4 to 512 and we ended up with successful results, in which minimum 90% percentage of boards ended up with zero conflicts.
As we already mentioned, Simulated Annealing is more dependent on temperature and cooling factor rather than problem size considering our aim to find successful results. For very large sizes of N-Queen, Simulated Annealing might not be optimal in terms of runtime, however for small and intermediate sizes Simulated Annealing shows promising results.

Group Members:

Barış Sevilmiş

Ufuk Akgeyik

Gökçe Sena Babacan

Imran Sharif Rizvi