# ITCS 6150 – Intelligent Systems Project Report

# Title: N-Queens Solution using Hill Climbing Variants


## Members: Amit Shetty, Sourav Roy Choudhury


## Date: 03-15-2019

**Purpose**:

The purpose of this project is to solve the N-Queens problem using 3 different variants of Hill Climbing Search. In order to explore the hill climbing approach, three techniques are discussed:

a. Hill Climbing using Steepest Ascent Approach
b. Hill Climbing using Steepest Ascent (using sideways move) Approach
c. Random Restart Hill Climbing Approach
d. Random Restart Hill Climbing (using sideways move) Approach

**Project Details:**

**Application Usage Overview:**

The application is designed to work as follows:

a. User is asked to enter the number of queens that they would like to test the local search techniques on. NOTE: To prevent the possibility of errors, validation has been added to ensure that the number of queens is more than 2 and the input should be a multiple of two.
b. User is then asked to input the hill climbing variant that they would like to test. The following options are given:
    a. Hill Climbing Steepest Ascent: In this approach, the application will calculate the best possible move based on a heuristic that checks the minimum number of times the queens can be attacked.
    b. Hill Climbing using Steepest Ascent (using sideways move): In this approach, the application will use the same logic as steepest ascent but this time if the heuristic gives the same result for the succeeding move, it will move sideways to solve the problem from a new state with the same heuristic cost. It will continue to do for a pre-determined amount of time, in case of this project it is set to 100 to get the best results.
    c. Random Restart Hill Climbing: In this approach, the application will use steepest hill climbing approach but every time it is not able to find a better heuristic in the succeeding move, the algorithm will start the approach again but this time form a completely random board with a new heuristic.
    d. Random Restart Hill Climbing (using sideways move): In this approach, the application will use steepest hill climbing approach but every time it finds an equal cost move it will attempt a sideways move i.e. attempt from a new board state with the same cost as the preceding move in the succeeding move. It will continue to do for a pre-determined amount of time, in case of this project it is set to 100 to get the best results.
c. User is then asked to enter the number of iterations they would like to try the approach for. User is given 8 choices from 300 to 1000 in increments of 100.
d. All these inputs are validated and the appropriate algorithm is called.
e. To make the output readable to the user, certain metrics are calculated which are as follows:
    a. Success Count: The total number of times the algorithm succeeded in getting a result based on the parameters it was given.
    b. Failure Count: The total number of times the algorithm failed in getting a result based on the parameters it was given.
    c. Initial State: Based on the problem statement, 4 random configurations are generated to test the algorithms on. The 4 initial random boards are shown.
    d. Final State: For every random state generated, of the algorithm succeeds in finding a solution, a final chess board is shown which is the solution for that initial random configuration using the algorithm defined by the user.

**Application Technical Details:**

Programming Language: Python

**Classes Used:**

1. nqueensapp.py:

Description:

- Is the starting point of execution for the program.
- Controls the flow of user input to their respective algorithms.
- Collects user input to define the type of algorithm needed to be implemented.
- Performs user input validation

Functions Used:

a. main():

Description:

- Is the main function of the entire application
- Collects the following user input:
    - Number of Queens: The number of queens to be placed on the board. To ensure there are no initial errors, validation is put in place for the input to be greater than two.
    - Type of Hill Climbing Algorithm: The user is asked to input their choice of algorithm which can be either Steepest Hill Climb with or without sideways movement.
    - Number of iterations: User is asked to input the number of iterations they want to attempt the algorithm where the choices range from 300 to 1000 in increments of 100.
- Contains the code for validating user input such as ensuring the number of queens is greater than 2 and valid choices are input by the user.

1. hillclimbingvariants.py:

Description:

- Contains the function calls for all the hill climbing variants implemented.
- Take the output and formats it to make it readable to the user.
- Calculates the metrics for success and failure.

Functions Used:

a. useSteepHillClimbingApproach():

Parameters Used:

- choiceofIteration: The number of times user has input for running the algorithm
- noofQueensOnBoard: The number of queens input by the user that are to be placed on the board.

Description:

- Calls the function to place the noofQueensOnBoard number of queens on the chessboard and perform the algorithm.

- Runs the algorithm for the specified choiceOfIteration and for a fixed four random initial configuration
- Calculates the success and failure metrics and presents it to the user alongside the initial random configuration and the final solution that has been achieved using steepest hill climb.

b. useSteepHillClimbingApporachWithSidewaysMove():

Parameters Used:

- choiceofIteration: The number of times user has input for running the algorithm
- noofQueensOnBoard: The number of queens input by the user that are to be placed on the board.

Description:

- Calls the function to place the noofQueensOnBoard number of queens on the chessboard and perform the algorithm.
- Runs the algorithm for the specified choiceOfIteration and for a fixed four random initial configuration
- Calculates the success and failure metrics and presents it to the user alongside the initial random configuration and the final solution that has been achieved using steepest hill climb with sideways move.

c. useRandomRestartHillClimbingApporach():

Parameters Used:

- choiceofIteration: The number of times user has input for running the algorithm
- noofQueensOnBoard: The number of queens input by the user that are to be placed on the board.

Description:

- Calls the function to place the noofQueensOnBoard number of queens on the chessboard and perform the algorithm.
- Runs the algorithm for the specified choiceOfIteration and for a fixed four random initial configuration
- Calculates the success and failure metrics and presents it to the user alongside the initial random configuration and the final solution that has been achieved using random restart hill climbing.

d. printFinalResult():

Parameters Used:

- localSearchAlgoUsed: String that tells which algorithm is used
- sample_sequences: The list containing the start and end states of the chessboard with the movement of the queens
- choiceofTries: Total number of times the algorithm will run as per the user
- total: The total number of times the algorithm will return a result

Description:

- Prints the metrics of the algorithm search in terms of the number of times it succeeded and failed.
- Prints the initial random state and final solution state of the puzzle.

3. hillclimbingalgos.py:

   Description:

- Contains the core logic for each of the hill climbing variants
- Calculates the heuristic of the current state of the board to determine the next best move.

Functions:

a. Steepest_ascent():

   Parameters Used:

- problem: The object containing the current state of the board and its corresponding heuristic cost among other parameters.
- allow_sideways: boolean value that determines if steepest hill climb must be attempted with or without sideways movement.
- max_sideways: numerical value that determines the maximum number of sideways movements allowed if sideways movement is enabled as per the user.

Description:

- Contains the core logic for using steepest ascent without sideways movement.
- Take the current state of the board and calculates the current heuristic and best possible move.
- If for every movement of the board there is a board with a better heuristic than the former then it will take use that option and place the queen on that board.
- Function returns a dictionary containing the solution of the random state it was processing as well as the path it took to calculate it.
- If allow_sideways is set to true then for every iteration where the succeeding move is equal to the preceding move in terms of heuristic values then the algorithm will perform a sideways movement I.e. try the steepest_ascent algorithm again but this time with a new board of the same heuristic.
- To avoid this happening for an indeterminate amount of time, a limit has been set in the max_sideways parameter. In case of this application to achieve the best result the algorithm limits the sideways moves to 100.

b. random_restart():

   Parameters Used:

- random_board: Object containing a random state of the board with all the queens in random locations
- noofQueensOnBoard: The numbers of queens defined by the user that have been placed on the board.

- allow_sideways: Boolean value that determines if sideways movement is allowed or not if same cost heuristic is encountered.

  Description:

- Random restart internally uses the same logic as steepest ascent algorithm with one difference.
- For every state in which the steepest ascent algorithm is unable to produce a result random restart will generate a new board and try again till it finds a result
- To limit the number of times it can do this a fixed limit of 100 has been added to get the best results.

c. get_next_best_move():

   Parameters Used:

- node: The board at its initial state
- problem: The object containing the current state of the board

  Description:

- Calculated the next best move based on the current state of the board.
- Best move is determined based on the lowest cost of the succeeding possible combinations based on the number of attacks possible by other queens.
- Functions returns the minimum of those values as the final result.

4. board_wrapper.py

   Description:

- Creates the current state of the board when invoked.
- The board is created based on the number of queens input by the user.
- Determine the cost in terms of how many queens can be attacked in the board's current state.
- Determines if the goal has been reached if the overall cost of the board returns zero.

  Functions:

a. __init__():

   Description:

- Role is to initialize the State object with the number of queens determined by the user.
- If the start state is not present then a new start state is generated for the board.

b. is_goal():

   Description:

- Determines if the goal node has been reached by checking if the number of queens that can be attacked is zero.

c. cost_function():

Description:

- Determines the cost function of the node by calculating the number of queens that can be attacked.

5. board.py

Description:

- Contains the bean of the current state of the board.
- Ensures a random board is generated for every new instantiation.
- Check is in place to ensure no duplicates are present in the board configurations.

Functions:

a. __init__():

Description:

- Initialises the class with the number of queens as the parameter
- Calculates the cost of the movement of the queens which is then used to determine the heuristic.

b. random_queen_position():

Description:

- Randomly generates a board and places queens on it.
- Takes the number of queens input by the user as a parameter.

c. get_children():

Description:

- For every position the queen is placed on the board, generate all possible combinations of the board.
- Returns the object containing a new random state of the board.

d. is_attack_possible ():

Description:

- Use the mathematical function to determine if the queens can attack each other based on straight and diagonal movements.
- Returns True or False based on the above fact.

e. calculate_possible_attacks():

Description:

- Based on the random queen positions generated, check each one against the is_attacking method and determine the pair of queens that attack each other.
- Returns the total length of the queen pairs that attack each other.

a. useRandomRestartHillClimbingApporachUsingSidewaysMove():

Parameters Used:

· choiceofIteration: The number of times user has input for running the algorithm
· noofQueensOnBoard: The number of queens input by the user that are to be placed on the board.

Description:

· Calls the function to place the noofQueensOnBoard number of queens on the chessboard and perform the algorithm with a sideways move
· Runs the algorithm for the specified choiceOfIteration and for a fixed four random initial configuration
· Calculates the success and failure metrics and presents it to the user alongside the initial random configuration and the final solution that has been achieved using random restart hill climbing.

**Sample Output**

**NOTE: Average Time for running the algorithm is between 30-60 seconds depending on the input combination given by the user.**

**1. Steepest Ascent Hill Climbing without Sideways Move**

/Users/amitshetty/.pyenv/versions/3.7.2/bin/python /Users/amitshetty/PycharmProjects/NQueensShort/nqueensapp.py

Enter the number of Queens to be placed on the board

8

Enter the choice of the algorithm to solve 8-queens problem

1. Steepest Hill Climb Algorithm

2. Steepest Hill Climb Using Sideways Move Algorithm

3. Random Restart Hill Climbing Algorithm

1

Enter number of times for the local search technique to be applied on the problem

1.300

2.400

3.500

4.600

5.700

6.800

7.900

8.1000

1

Steepest Ascent Hill Climbing Results :

Success Count : 28

Failure : 272

Random Initial Configuration #1.

Initial State :

_ _ _ _ _ Q _ _

_ _ _ Q _ _ _ _

_ _ _ _ _ _ _ _

_ _ _ _ Q _ _ Q

_ _ _ _ _ _ _ _

_ Q _ _ _ _ _ _

_ _ _ _ _ _ Q _

Q _ Q _ _ _ _ _

Final State :

_ _ _ _ _ Q _ _

_ _ _ Q _ _ _ _

Q _ _ _ _ _ _ _

_ _ _ _ Q _ _ _

_ _ _ _ _ _ _ Q

_ Q _ _ _ _ _ _

_ _ _ _ _ _ Q _

_ _ Q _ _ _ _ _

Random Initial Configuration #2.

Initial State :

Q _ _ _ _ Q Q _

_ _ _ _ _ _ _ _

_ _ Q _ Q _ _ _

_ _ _ Q _ _ _ Q

_ _ _ _ _ _ _ _

_ _ _ _ _ _ _ _

_ Q _ _ _ _ _ _

_ _ _ _ _ _ _ _

Final State :

_ _ _ _ _ Q _ _

_ _ Q _ _ _ _ _

_ _ _ _ Q _ _ _

_ _ _ _ _ _ _ Q

Q _ _ _ _ _ _ _

_ _ _ Q _ _ _ _

_ Q _ _ _ _ _ _

_ _ _ _ _ _ Q _

Random Initial Configuration #3.

Initial State :

_ _ _ _ _ _ _ _

_ Q _ _ _ _ _ Q

_ _ _ _ _ _ Q _

Q _ _ Q Q _ _ _

_ _ _ _ _ _ _ _

_ _ _ _ _ Q _ _

_ _ _ _ _ _ _ _

_ _ Q _ _ _ _ _

Final State :

_ _ _ _ _ Q _ _

_ _ _ Q _ _ _ _

_ _ _ _ _ _ Q _

Q _ _ _ _ _ _ _

_ _ _ _ _ _ _ Q

_ Q _ _ _ _ _ _

_ _ _ _ Q _ _ _

_ _ Q _ _ _ _ _

Random Initial Configuration #4.

Initial State :

_ _ Q Q _ _ _ _

_ _ _ _ _ _ _ _

_ _ _ _ _ _ _ _

Q _ _ _ _ _ _ _

_ _ _ _ Q Q _ _

_ _ _ _ _ _ _ _

_ Q _ _ _ _ Q _

_ _ _ _ _ _ _ Q

Final State :

_ _ Q _ _ _ _ _

_ _ _ _ _ Q _ _

_ _ _ _ _ _ _ Q

Q _ _ _ _ _ _ _

_ _ _ _ Q _ _ _

_ _ _ _ _ _ Q _

_ Q _ _ _ _ _ _

_ _ _ Q _ _ _ _

Process finished with exit code 0

**2. Steepest Ascent Hill Climbing with Sideways Move**

/Users/amitshetty/.pyenv/versions/3.7.2/bin/python
/Users/amitshetty/PycharmProjects/NQueensShort/nqueensapp.py

Enter the number of Queens to be placed on the board

8

Enter the choice of the algorithm to solve 8-queens problem

1. Steepest Hill Climb Algorithm

2. Steepest Hill Climb Using Sideways Move Algorithm

3. Random Restart Hill Climbing Algorithm

2

Enter number of times for the local search technique to be applied on the problem

1.300

2.400

3.500

4.600

5.700

6.800

7.900

8.1000

4

Steepest Ascent Hill Climbing with Sideway Move Results :

Success Count : 569

Failure : 31

Random Initial Configuration #1.

Initial State :

_ _ _ _ _ _ _ _

_ _ _ _ _ Q _ _

_ _ _ _ _ _ _ Q

_ _ Q Q Q _ _ _

_ Q _ _ _ _ _ _

Q _ _ _ _ _ Q _

_ _ _ _ _ _ _ _

_ _ _ _ _ _ _ _

Final State :

_ _ Q _ _ _ _ _

_ _ _ _ _ Q _ _

_ Q _ _ _ _ _ _

_ _ _ _ Q _ _ _

_ _ _ _ _ _ _ Q

Q _ _ _ _ _ _ _

_ _ _ _ _ _ Q _

_ _ _ Q _ _ _ _

Random Initial Configuration #2.

Initial State :

_ _ _ Q _ _ _ _

_ _ Q _ Q _ _ _

_ _ _ _ _ _ _ _

_ Q _ _ _ _ _ _

_ _ _ _ _ _ _ _

_ _ _ _ _ Q _ _

_ _ _ _ _ _ _ _

Q _ _ _ _ _ Q Q

Final State :

_ _ Q _ _ _ _ _

_ _ _ _ _ Q _ _

_ _ _ _ _ _ _ Q

Q _ _ _ _ _ _ _

_ _ _ _ Q _ _ _

_ _ _ _ _ _ Q _

_ Q _ _ _ _ _ _

_ _ _ Q _ _ _ _

Random Initial Configuration #3.

Initial State :

_ _ _ _ _ _ _ _

_ _ _ _ _ Q _ _

Q _ _ _ _ _ Q _

_ Q _ Q _ _ _ _

_ _ _ _ Q _ _ Q

_ _ _ _ _ _ _ _

_ _ Q _ _ _ _ _

_ _ _ _ _ _ _ _

Final State :

_ _ _ _ _ Q _ _

_ _ Q _ _ _ _ _

_ _ _ _ _ _ Q _

_ _ _ Q _ _ _ _

Q _ _ _ _ _ _ _

_ _ _ _ _ _ _ Q

_ Q _ _ _ _ _ _

_ _ _ _ Q _ _ _

Random Initial Configuration #4.

Initial State :

_ _ _ _ _ _ _ _

_ _ _ Q _ Q _ _

_ _ _ _ _ _ _ Q

_ _ _ _ _ _ Q _

Q Q _ _ Q _ _ _

_ _ _ _ _ _ _ _

_ _ Q _ _ _ _ _

_ _ _ _ _ _ _ _

Final State :

_ _ _ Q _ _ _ _

_ _ _ _ _ Q _ _

_ _ _ _ _ _ _ Q

_ _ Q _ _ _ _ _

Q _ _ _ _ _ _ _

_ _ _ _ _ _ Q _

_ _ _ _ Q _ _ _

_ Q _ _ _ _ _ _

Process finished with exit code 0

**3. Random Restart Hill Climbing**

/Users/amitshetty/.pyenv/versions/3.7.2/bin/python
/Users/amitshetty/PycharmProjects/NQueensShort/nqueensapp.py

Enter the number of Queens to be placed on the board

8

Enter the choice of the algorithm to solve 8-queens problem

1. Steepest Hill Climb Algorithm

2. Steepest Hill Climb Using Sideways Move Algorithm

3. Random Restart Hill Climbing Algorithm

3

Enter number of times for the local search technique to be applied on the problem

1.300

2.400

3.500

4.600

5.700

6.800

7.900

8.1000

1

Steepest Ascent with Random Restart Hill Climbing Results :

Success Count : 300

Failure : 0

Random Initial Configuration #1.

Initial State :

_ Q _ _ _ Q _ _

_ _ _ _ _ _ _ _

Q _ _ Q _ _ _ _

_ _ Q _ _ _ _ _

_ _ _ _ Q _ Q _

_ _ _ _ _ _ _ _

_ _ _ _ _ _ _ _

_ _ _ _ _ _ _ Q

Final State :

_ _ _ _ _ _ Q _

_ _ _ Q _ _ _ _

_ Q _ _ _ _ _ _

_ _ _ _ Q _ _ _

_ _ _ _ _ _ _ Q

Q _ _ _ _ _ _ _

_ _ Q _ _ _ _ _

_ _ _ _ _ Q _ _

Random Initial Configuration #2.

Initial State :

Q _ _ _ _ _ _ _

_ _ _ _ _ _ _ _

_ _ _ _ _ _ Q _

_ _ Q _ _ Q _ Q

_ Q _ Q Q _ _ _

_ _ _ _ _ _ _ _

_ _ _ _ _ _ _ _

_ _ _ _ _ _ _ _

Final State :

_ _ Q _ _ _ _ _

Q _ _ _ _ _ _ _

_ _ _ _ _ _ Q _

_ _ _ _ Q _ _ _

_ _ _ _ _ _ _ Q

_ Q _ _ _ _ _ _

_ _ _ Q _ _ _ _

_ _ _ _ _ Q _ _

Random Initial Configuration #3.

Initial State :

_ Q _ Q _ _ _ _

Q _ _ _ _ Q _ _

_ _ _ _ Q _ _ _

_ _ _ _ _ _ _ Q

_ _ _ _ _ _ _ _

_ _ _ _ _ _ Q _

_ _ Q _ _ _ _ _

_ _ _ _ _ _ _ _

Final State :

_ _ _ Q _ _ _ _

Q _ _ _ _ _ _ _

_ _ _ _ Q _ _ _

_ _ _ _ _ _ _ Q

_ Q _ _ _ _ _ _

_ _ _ _ _ _ Q _

_ _ Q _ _ _ _ _

_ _ _ _ _ Q _ _

Random Initial Configuration #4.

Initial State :

_ _ _ _ _ _ _ _

_ _ Q _ _ _ _ _

_ Q _ _ _ _ _ _

_ _ _ _ _ _ Q _

_ _ _ _ _ Q _ _

_ _ _ _ _ _ _ _

_ _ _ _ Q _ _ _

Q _ _ Q _ _ _ Q

Final State :

_ _ _ Q _ _ _ _

_ _ _ _ _ _ _ Q

Q _ _ _ _ _ _ _

_ _ _ _ Q _ _ _

_ _ _ _ _ _ Q _

_ Q _ _ _ _ _ _

_ _ _ _ _ Q _ _

_ _ Q _ _ _ _ _

Process finished with exit code 0

4. Random Restart Hill Climbing with Sideways Move

/Users/amitshetty/.pyenv/versions/3.7.2/bin/python
/Users/amitshetty/PycharmProjects/NQueensShort/nqueensapp.py

Enter the number of Queens to be placed on the board

8

Enter the choice of the algorithm to solve 8-queens problem

1. Steepest Hill Climb Algorithm

2. Steepest Hill Climb Using Sideways Move Algorithm

3. Random Restart Hill Climbing Algorithm

4. Random Restart Hill Climbing Algorithm using sideways motion

4

Enter number of times for the local search technique to be applied on the problem

1.300

2.400

3.500

4.600

5.700

6.800

7.900

8.1000

4

Steepest Ascent with Random Restart Hill Climbing with Sideways Move Results :

Success Count : 600

Failure : 0

Random Initial Configuration #1.

Initial State :

_ _ Q _ _ _ _ _

Q _ _ _ Q _ _ _

_ _ _ _ _ _ _ _

_ Q _ _ _ _ _ _

_ _ _ Q _ _ _ Q

_ _ _ _ _ Q _ _

_ _ _ _ _ _ Q _

_ _ _ _ _ _ _ _

Final State :

_ _ _ _ Q _ _ _

_ Q _ _ _ _ _ _

_ _ _ _ _ _ _ Q

Q _ _ _ _ _ _ _

_ _ _ Q _ _ _ _

_ _ _ _ _ _ Q _

_ _ Q _ _ _ _ _

_ _ _ _ _ Q _ _

Random Initial Configuration #2.

Initial State :

_ _ _ _ _ Q _ _

_ _ _ _ _ _ _ _

_ _ _ _ _ _ _ _

_ Q _ _ _ _ _ _

_ _ Q Q Q _ _ _

_ _ _ _ _ _ Q Q

Q _ _ _ _ _ _ _

_ _ _ _ _ _ _ _

Final State :

_ _ _ _ _ Q _ _

_ _ Q _ _ _ _ _

_ _ _ _ _ _ Q _

_ Q _ _ _ _ _ _

_ _ _ _ _ _ _ Q

_ _ _ _ Q _ _ _

Q _ _ _ _ _ _ _

_ _ _ Q _ _ _ _

Random Initial Configuration #3.

Initial State :

_ _ _ _ _ _ _ _

_ _ _ _ _ Q _ _

_ _ _ _ _ _ Q Q

_ _ Q _ _ _ _ _

_ _ _ _ _ _ _ _

_ Q _ _ _ _ _ _

_ _ _ Q _ _ _ _

Q _ _ _ Q _ _ _

Final State :

_ _ _ Q _ _ _ _

_ _ _ _ _ Q _ _

_ _ _ _ _ _ _ Q

_ Q _ _ _ _ _ _

_ _ _ _ _ _ Q _

Q _ _ _ _ _ _ _

_ _ Q _ _ _ _ _

_ _ _ _ Q _ _ _

Random Initial Configuration #4.

Initial State :

_ _ _ _ _ _ _ _

_ Q _ _ _ Q _ _

Q _ _ _ _ _ Q _

_ _ _ _ _ _ _ _

_ _ _ _ _ _ _ Q

_ _ _ _ _ _ _ _

_ _ _ Q _ _ _ _

_ _ Q _ Q _ _ _

Final State :

_ _ _ _ _ _ Q _

Q _ _ _ _ _ _ _

_ _ Q _ _ _ _ _

_ _ _ _ _ _ _ Q

_ _ _ _ _ Q _ _

_ _ _ Q _ _ _ _

_ Q _ _ _ _ _ _

_ _ _ _ Q _ _ _

Process finished with exit code 0

**Source Code:**

Nqueensapp.py:

```
import sys
import hillclimbingvariants

def main():
    # Default condition
    noofQueensOnBoard = int(input('Enter the number of Queens to be placed on the board\n') or
'4')
    # Logical check
    if noofQueensOnBoard <= 2 or noofQueensOnBoard % 2 != 0 :
        print('Invalid Input. Please enter a value greater than and a multiple of 2')
        sys.exit(1)
    # User input for choice of algorithm
    print('Enter the choice of the algorithm to solve {}-queens
problem'.format(noofQueensOnBoard))
    choiceOfAlgorithm = int(input('1. Steepest Hill Climb Algorithm\n2. Steepest Hill Climb Using
Sideways Move Algorithm\n3. Random Restart Hill Climbing Algorithm\n4. Random Restart
Hill Climbing Algorithm using sideways motion\n') or '1')
    print('Enter number of times for the local search technique to be applied on the problem')
    choiceOfTries = int(input('1.300\n2.400\n3.500\n4.600\n5.700\n6.800\n7.900\n8.1000\n') or '1')
    if choiceOfTries == 1:
        numOfIteration = 300
```

```
    elif choiceOfTries == 2:
        numOfIteration = 400
    elif choiceOfTries == 3:
        numOfIteration = 500
    elif choiceOfTries == 4:
        numOfIteration = 600
    elif choiceOfTries == 5:
        numOfIteration = 700
    elif choiceOfTries == 6:
        numOfIteration = 800
    elif choiceOfTries == 7:
        numOfIteration = 900
    elif choiceOfTries == 8:
        numOfIteration = 1000
    else :
        print('Invalid Choice for number of tries. Please try again')
        sys.exit(1)
    if choiceOfAlgorithm == 1:
        hillclimbingvariants.useSteepHillClimbingApproach(numOfIteration,noofQueensOnBoard)
    elif choiceOfAlgorithm == 2:

hillclimbingvariants.useSteepHillClimbingApporachWithSidewaysMove(numOfIteration,noofQueens
OnBoard)
    elif choiceOfAlgorithm == 3:

hillclimbingvariants.useRandomRestartHillClimbingApporach(numOfIteration,noofQueensOnBoard)
    elif choiceOfAlgorithm == 4:

hillclimbingvariants.useRandomRestartHillClimbingApporachWithSidewaysMove(numOfIteration,n
oofQueensOnBoard)
    else :
        print('Invalid Entry for Algorithm Choice. Try again')
        sys.exit(1)


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

hillclimbingvariants.py

```
import hillclimbingalgos
import boardwrapper

# Steepest Ascent without sideway move method
def useSteepHillClimbingApproach(choiceOfIteration,noofQueensOnBoard):
    total = 0
    fail_steps = 0
    solution_path = []
```

```
    generated_boards = []
    for _ in range(choiceOfIteration):
        final_state =
hillclimbingalgos.steepest_ascent(boardwrapper.BoardWrapper(noofQueensOnBoard))
        total += final_state['is_final_state']
        fail_steps += len(final_state['solution'])
        if final_state['is_final_state']:
            if (final_state['problem'] not in generated_boards) and (len(generated_boards) < 4):
                generated_boards.append(final_state['problem'])
                solution_path.append(final_state['solution'])
    printFinalResult('Steepest Ascent Hill Climbing', solution_path, choiceOfIteration, total)

# Steepest Ascent with Sideway move up to 100 moves
def useSteepHillClimbingApporachWithSidewaysMove(choiceOfIteration,noofQueensOnBoard):
    total = 0
    fail_steps = 0
    solution_path = []
    generated_boards = []
    for _ in range(choiceOfIteration):
        final_state =
hillclimbingalgos.steepest_ascent(boardwrapper.BoardWrapper(noofQueensOnBoard),
allow_sideways=True)
        total += final_state['is_final_state']
        fail_steps += len(final_state['solution'])
        if final_state['is_final_state']:
            if (final_state['problem'] not in generated_boards) and (len(generated_boards) < 4):
                generated_boards.append(final_state['problem'])
                solution_path.append(final_state['solution'])
    printFinalResult('Steepest Ascent Hill Climbing with Sideway Move', solution_path,
choiceOfIteration, total)

# Steepest Ascent with Random Restart (no sideway movement)
def useRandomRestartHillClimbingApporach(choiceOfIteration,noofQueensOnBoard):
    total = 0
    fail_steps = 0
    solution_path = []
    generated_boards = []
    for _ in range(choiceOfIteration):
        final_state =
hillclimbingalgos.random_restart(boardwrapper.BoardWrapper(noofQueensOnBoard).__class__,
noofQueensOnBoard, allow_sideways=False)
        total += final_state['is_final_state']
        fail_steps += len(final_state['solution'])
        if final_state['is_final_state']:
            if (final_state['problem'] not in generated_boards) and (len(generated_boards) < 4):
                generated_boards.append(final_state['problem'])
                solution_path.append(final_state['solution'])
    printFinalResult('Steepest Ascent with Random Restart Hill Climbing', solution_path,
choiceOfIteration, total)

# Steepest Ascent with Random Restart (no sideway movement)
```

```python
def
useRandomRestartHillClimbingApporachWithSidewaysMove(choiceOfIteration,noofQueensOnBoar
d):
    total = 0
    fail_steps = 0
    solution_path = []
    generated_boards = []
    for _ in range(choiceOfIteration):
        final_state =
hillclimbingalgos.random_restart(boardwrapper.BoardWrapper(noofQueensOnBoard).__class__,
noofQueensOnBoard, allow_sideways=True)
        total += final_state['is_final_state']
        fail_steps += len(final_state['solution'])
        if final_state['is_final_state']:
            if (final_state['problem'] not in generated_boards) and (len(generated_boards) < 4):
                generated_boards.append(final_state['problem'])
                solution_path.append(final_state['solution'])
    printFinalResult('Steepest Ascent with Random Restart Hill Climbing with Sideways Move',
solution_path, choiceOfIteration, total)

def printFinalResult(localSearchAlgoUsed, sample_sequences, choiceOfTries, total):
    print('{} Results :\nSuccess Count : {}\nFailure : {}'.format(localSearchAlgoUsed, total,
choiceOfTries - total))
    for i, currentState in enumerate(sample_sequences):
        print('Random Initial Configuration #{}.'.format(i + 1))
        print('Initial State :\n{}'.format(currentState[0]))
        print('Final State :\n{}'.format(currentState[-1]))
```

hillclimbingalgos.py:

```python
import random

# Steepest ascent with and without sideways
def steepest_ascent(board_wrapper, allow_sideways=False, max_sideways=100):
    node = board_wrapper.start_state
    node_cost = board_wrapper.cost_function(node)
    path = []
    sideways_moves = 0
    while True:
        path.append(node)
        best_move = get_next_best_move(node, board_wrapper)
        best_move_cost = board_wrapper.cost_function(best_move)
        if best_move_cost > node_cost:
            break
        elif best_move_cost == node_cost:
            if not allow_sideways or sideways_moves == max_sideways:
                break
            else:
```

```
            sideways_moves += 1
        else:
            sideways_moves = 0
        node = best_move
        node_cost = best_move_cost
    return {'is_final_state': 1 if board_wrapper.is_goal(node) else 0, 'solution': path , 'problem':
board_wrapper}

# Random restart using steepest ascent with and without sideways
def random_restart(random_board, noofQueensOnBoard, allow_sideways):
    num_restarts = 100
    path = []
    for _ in range(num_restarts):
        result = steepest_ascent(random_board(noofQueensOnBoard), allow_sideways)
        path += result['solution']
        if result['is_final_state'] == 1:
            break
    result['solution'] = path
    return result

# Calculate the cost of getting the nest best move based on the queen attack heuristic
def get_next_best_move(node, problem):
    best_moves = node.get_children()
    moves_cost = [problem.cost_function(child) for child in best_moves]
    min_cost = min(moves_cost)
    best_move = random.choice([move for move_index, move in enumerate(best_moves) if
moves_cost[move_index] == min_cost])
    return best_move
```

boardwrapper.py:

```
import board

# Wrapper class that holds the current state of the board object
class BoardWrapper:

    def __init__(self, noofQueensOnBoard, start_state=None):
        if not start_state:
            start_state = board.Board(noofQueensOnBoard)
        self.start_state = start_state

    def is_goal(self, state):
        # Check goal
        return state.calculate_possible_attacks() == 0

    def cost_function(self, state):
        # Cost function as number of queen attacking
        return state.calculate_possible_attacks()
```

board.py:

```
import random
import copy

# Board object containing the number of queens on the board and their current positions.
# For every board object generated child states for each queen movement and their heurisitc value is calculated
class Board:
    count = 0
    def __init__(self, numberOfQueens, queen_positions=None, parent=None, move_cost=0):
        if queen_positions is None:
            self.queen_num = numberOfQueens
            self.queen_positions = frozenset(self.random_queen_position())
        else:
            self.queen_positions = queen_positions
            self.queen_num = len(self.queen_positions)
        self.f_cost = move_cost
        self.parent = parent
        self.id = Board.count
        Board.count += 1

    def __str__(self):
        # string function for printing
        return '\n'.join([' '.join(['_' if (col, row) not in self.queen_positions else 'Q' for col in range(self.queen_num)]) for row in range(self.queen_num)])

    def __hash__(self):
        # Hash function to remove duplicate
        return hash(self.queen_positions)

    def __eq__(self, other):
        # Check if 2 nodes are same
        return self.queen_positions == other.queen_positions

    def __lt__(self, other):
        # Compare cost
        return self.f_cost < other.f_cost or (self.f_cost == other.f_cost and self.id > other.id)

    def random_queen_position(self):
        # Generate random queen position
        open_columns = list(range(self.queen_num))
        queen_positions = [(open_columns.pop(random.randrange(len(open_columns))), random.randrange(self.queen_num)) for _ in range(self.queen_num)]
        return queen_positions

    def get_children(self):
        # Get children from current state node
        children = []
        parent_queen_positions = list(self.queen_positions)
```

```
        for queen_index, queen in enumerate(parent_queen_positions):
            new_positions = [(queen[0], row) for row in range(self.queen_num) if row != queen[1]]
            for new_position in new_positions:
                queen_positions = copy.deepcopy(parent_queen_positions)
                queen_positions[queen_index] = new_position
                children.append(Board(self.queen_num, queen_positions))
        return children

    def random_child(self):
        # Random child
        queen_positions = list(self.queen_positions)
        random_queen_index = random.randrange(len(self.queen_positions))
        queen_positions[random_queen_index] = (queen_positions[random_queen_index][0],
random.choice([row for row in range(self.queen_num) if row !=
queen_positions[random_queen_index][1]]))
        return Board(self.queen_num, queen_positions)

    def range_between(self, a, b):
        # Return positions between a and b
        if a > b:
            return range(a-1, b, -1)
        elif a < b:
            return range(a+1, b)
        else:
            return [a]

    def consolidate_attack_moves(self, a, b):
        # Repeat
        if len(a) == 1:
            a *= len(b)
        elif len(b) == 1:
            b *= len(a)
        return zip(a, b)

    # Repeat zipped positions between a and b
    def attack_positions(self, a, b):
        return self.consolidate_attack_moves(list(self.range_between(a[0], b[0])),
list(self.range_between(a[1], b[1])))

    # Check if 2 positions have attacked each other
    def is_attack_possible(self, queens, a, b):
        if (a[0] == b[0]) or (a[1] == b[1]) or (abs(a[0]-b[0]) == abs(a[1]-b[1])):
            for between in self.attack_positions(a, b):
                if between in queens:
                    return False
            return True
        return False

    # Calculate number of queen pairs attacking each other
    def calculate_possible_attacks(self):
        attacking_pairs = []
```

```
queen_positions = list(self.queen_positions)
left_to_check = copy.deepcopy(queen_positions)
while left_to_check:
    a = left_to_check.pop()
    for b in left_to_check:
        if self.is_attack_possible(queen_positions, a, b):
            attacking_pairs.append([a, b])
return len(attacking_pairs)
```