Intelligent Systems Project-2 Report

Solving N-queens problem using hill-climbing and its variants.

Team Members:

Shreya Vinodh	**801149383**
Utkarsha Gurjar	**801149356**
Afreen	**801136632**

Hill-Climbing Search:

The hill-climbing search algorithm is a loop that continually moves in the direction of the best successor value and terminates when there are no neighboring states with better values. The hill climbing algorithm usually gets stuck because of local maxima, ridges and plateaus.

In case of 8-queens problem, hill climbing gets stuck around 86% of the time and solves about 14% of problem instances. It takes 4 steps on average when it succeeds and 3 when it gets stuck.

To improve the results when stuck on a plateau, sideways move is implemented to check if the plateau is a shoulder i.e. to keep moving forward on the plateau state; however, an infinite loop can occur when the algorithm reaches a flat local maximum/minimum that is not a shoulder. To solve this problem, a limit of say 100 consecutive sideways moves can be implemented.

This raises the percentage of problem instances solved by hill climbing from 14% to 94%. With this addition, the algorithm averages roughly 21 steps for each successful instance and 64 for each failure.

The hill-climbing algorithms are incomplete—they often fail to find a goal when one exists because they can get stuck on local maxima. Random-restart hill climbing conducts a series of hill-climbing searches from randomly generated initial states, until a goal is found.

This implementation completes with probability approaching 1, because it will eventually generate a goal state as the initial state. If each hill-climbing search has a probability p of success, then the expected number of restarts required is $1/p$. For 8-queens instances with no sideways moves allowed, $p \approx 0.14$, so we need roughly 7 iterations to find a goal (6 failures and 1 success). The expected number of steps is the cost of one successful iteration plus $(1-p)/p$

times the cost of failure, or roughly 22 steps in all. When we allow sideways moves, 1/0.94 ≈ 1.06 iterations are needed on average and (1×21)+(0.06/0.94)×64 ≈ 25 steps.

n-Queens Formulation:

Initial state:
An n*n matrix with n queens placed randomly one per row in the matrix. For eg. 8*8 matrix with 8 queens placed randomly one per row in the matrix.

Heuristic:
The heuristic cost h, is the number of pairs of queens that are attacking each other, either directly or indirectly.

Goal state:
An n*n matrix where each queen is placed on one position per row, such that no queen attacks another either horizontally, vertically or diagonally.

Goal Test :
When the heuristic h of a state is zero, that state is considered to be the goal.

Successor function:
Each successor is obtained by adding one queen in an empty square

Program Structure:

Create an object of class board start
Class board initializes variables for calculating the total number of steps, successes, failures etc.

start calls main function defined in class board main()
main() function asks the user for the number of queens to be placed on the board, and the number of runs to be executed.
After getting these inputs, implement hill-climbing algorithm. calc_board()
Implement hill-climbing algorithm with sideways move. calc_board_wSideway()
Implement Random-Restart algorithm without sideways move and with sideways move. calc_RandomRestart()

Class state initializes the size of the board, i.e. no. of rows and no.of columns (same as the no. of queens to be placed on the board), ‘h’ value, and data to a board state.
This class has functions to:
create a random initial board state generate_initial()
generate successor board state generate_successor()

calculate h value for each board state calc_h()

Global variables used:

n – Number of queens to be placed on the board; input by the user.
h_dash – A variable that stores the minimum heuristic value of a state of the board.

Functions/Procedure used to compute Heuristic Function:

calc_h(): counts the number of queens that are placed on the attacking positions from a particular position. A helper function findAttckPos() is used to find the attacking positions, i.e vertically and diagonally.

findAttckPos(): finds the attacking positions from a particular position on the board. Returns two lists - one with vertical attack positions and another with diagonal attack positions.

Other functions/procedures used:

Apart from the above mentioned heuristic functions, there are 2 constructors and 8 methods:

1. The constructor of class “state” initializes the size of the board, board data containing the queens and the blank spaces.
2. The function “generate_initial” generates the board randomly.
3. The function “generate_successor” generates the successor board.
4. The helper function “copy” is used to copy the parent board in order to generate the successor board.
5. The constructor of class “board” initializes the success steps, failure steps, total no.of steps, total success steps, total failure steps.
6. The function “find_rowPos” is used to find the position of the queen of a particular row on the initial board.
7. The function “main” asks the user for the number of queens to be placed on the board, and the number of runs to be executed.
8. The function “calc_board” performs hill climbing algorithm.
9. The function “calc_board_wSideway” performs hill climbing with sideways move.
10. The function “calc_RandomRestart” performs hill climbing with random restart method.

Program Code:

```
import random
import math

class state:
#   global variable to store the h value of each board state.
    global h_dash

#   constructor of class state that initializes size, data, and h field of the board.
    def __init__(self, size, data=None):
        self.size = size
        self.data = data
        self.h = -1                                 #initializing with a non-positive number

#   function that generates the initial configuration of the board.
    def generate_initial(self):
        board_s=[]
        for i in range(int(self.size)):
            col=[None]*int(self.size)
            rand = random.randint(0,int(self.size)-1)
            for j in range(int(self.size)):
                if j == rand:
                    col[j]='Q'
                else:
                    col[j]='_'
            board_s.append(col)
        self.data = board_s

        return self.data

#   function to generate a successor board by moving the queen from its current
location(old_pos) to a
#   specified location(new_pos) on the board.
    def generate_successor(self, old_pos, new_pos):
        x = new_pos[0]
        y = new_pos[1]

        old_x = old_pos[0]
```

```
        old_y = old_pos[1]

        nxt = self.copy(self.data)

        for i in range(0, len(nxt)):
            for j in range(0, len(nxt)):
                if i==x and j==y:
                    nxt[i][j]='Q'

                if i==old_x and j==old_y:
                    nxt[i][j]='_'
        return nxt

#   helper function to copy the data of the board of which a successor needs to be generated,
#   onto a new board.
    def copy(self,prev):
        copy_board=[]

        for i in prev:
            squares=[]
            for j in i:
                squares.append(j)
            copy_board.append(squares)
        return copy_board

#   function to calculate the h value of a board.
    def calc_h(self):
        count=0

        for i in range(len(self.data)):
            for j in range(len(self.data)):
                if self.data[i][j]=='Q':
                    pos = [None]*2
                    pos[0]=i
                    pos[1]=j
                    v,d = self.findAttckPos(pos)

                    for k in range(len(self.data)):
                        for l in range(len(self.data)):
```

```
                  if self.data[k][l]=='Q':
                     for e in v:
                        if e[0] == k and e[1] == l:
                           count+=1
                     for e in d:
                        if e[0] == k and e[1] == l:
                           count+=1
      return math.ceil(count/2)

#   helper function to calculate the attack positions(vertical and diagonal) from a
#   particular position(x,y) on the board.
   def findAttckPos(self,pos):
      ver = []                    #an empty list to store the vertical attack positions
      x=pos[0]
      y=pos[1]

      if x==0:
         while(x < len(self.data)-1):
            attck_v=[None]*2
            x+=1

            attck_v[0]=x
            attck_v[1]=y
            ver.append(attck_v)

      elif x==len(self.data):
         while(x>=0):
            attck_v=[None]*2
            x-=1

            attck_v[0]=x
            attck_v[1]=y
            ver.append(attck_v)

      else:
         while(x < len(self.data)-1):
            attck_v=[None]*2
            x+=1
```

```
            attck_v[0]=x
            attck_v[1]=y
            ver.append(attck_v)

        x=pos[0]
        y=pos[1]

        while(x>=0):
            attck_v=[None]*2
            x-=1

            attck_v[0]=x
            attck_v[1]=y
            ver.append(attck_v)

    diag = []                    #an empty list to store the diagonal attack positions
    x=pos[0]
    y=pos[1]
    #1. Checking upper left squares
    while x>0 and x<len(self.data) and y>0 and y<len(self.data):
        attck_d=[None]*2
        x-=1
        y-=1

        attck_d[0]=x
        attck_d[1]=y
        diag.append(attck_d)

    x=pos[0]
    y=pos[1]
    #2. Checking lower left squares
    while x>=0 and x<len(self.data)-1 and y>0 and y<len(self.data):
        attck_d=[None]*2
        x+=1
        y-=1

        attck_d[0]=x
        attck_d[1]=y
        diag.append(attck_d)
```

```
        x=pos[0]
        y=pos[1]
        #3. Checking upper right squares
        while x>0 and x<len(self.data) and y>=0 and y<len(self.data)-1:
          attck_d=[None]*2
          x-=1
          y+=1

          attck_d[0]=x
          attck_d[1]=y
          diag.append(attck_d)

        x=pos[0]
        y=pos[1]
        #4. Checking lower right squares
        while x>=0 and x<len(self.data)-1 and y>=0 and y<len(self.data)-1:
          attck_d=[None]*2
          x+=1
          y+=1

          attck_d[0]=x
          attck_d[1]=y
          diag.append(attck_d)

        return ver, diag

class board:
#   global variable that stores the input of number of queens to be placed on the board from a
user.
    global n

    def __init__(self):
        self.successes = 0
        self.failures = 0
        self.steps = 0
        self.tot_sSteps = 0
        self.tot_fSteps = 0
        self.count_init = 0
```

```
#   function to find the position of the queen on a particular row on the initial board.
    def find_rowPos(self,data,row_num):
        for j in range(len(data)):
            if data[row_num][j] == 'Q':
                pos = [None]*2
                pos[0]= row_num
                pos[1]= j
                break
        return pos

#   main function
    def main(self):
        global n

        print("How many Queens do you want to place on the board?")
        n = input()

        print("How many runs?")
        r = input()

        print()

#       Call for Hill-Climbing search.
        print("Hill-Climbing search:")
        for x in range(int(r)):
            self.steps = 0
            self.calc_board(x)
        print()
        print("....")
        print("....")
        print("....")
        print("....")
        print("....")
        print()
        print("Hill-Climbing search stats for " + str(n) + "-Queens problem for " + str(r) + " runs:")
        print("Success Rate: ", 100*self.successes/int(r),"%")
        print("Failure Rate: ", 100*self.failures/int(r), "%")
        print("Average number of steps for successes: ", self.tot_sSteps/self.successes)
```

```
        print("Average number of steps for failures: ", self.tot_fSteps/self.failures)
        print()

#       call for Hill-climbing search with sideways move.
        print("Hill-Climbing search with sideways move:")
        self.successes = 0
        self.failures = 0
        self.steps = 0
        self.tot_sSteps = 0
        self.tot_fSteps = 0

        for x in range(int(r)):
            self.steps = 0
            self.calc_board_wSideway(x)
        print()
        print("....")
        print("....")
        print("....")
        print("....")
        print("....")
        print()
        print("Hill-Climbing search with sideways move stats for " + str(n) + "-Queens problem for "
+ str(r) + " runs:")
        print("Success Rate: ", 100*self.successes/int(r),"%")
        print("Failure Rate: ", 100*self.failures/int(r), "%")
        print("Average number of steps for successes: ", self.tot_sSteps/self.successes)
        print("Average number of steps for failures: ", self.tot_fSteps/self.failures)
        print()

#       call for random-restart without sideways move.
        print("Random-Restart Hill-Climbing search:")
        self.steps = 0
        self.count_init = 0
        for x in range(int(r)):
            y=self.steps
            self.calc_RandomRestart(0)
        print()
        print("Random-Restart Hill-Climbing search without sideways move stats for " + str(n) +
"-Queens problem for " + str(r) + " runs:")
```

```
        print("Average number of random-restarts without sideways move: ", self.count_init/int(r))
        print("Average number of steps required without sideways move: ", self.steps/int(r))
        print()

#       call for random-restart with sideways move.
        self.steps = 0
        self.count_init = 0
        for x in range(int(r)):
            self.calc_RandomRestart(1)
        print()
        print("Random-Restart Hill-Climbing search with sideways move stats for " + str(n) +
"-Queens problem for " + str(r) + " runs:")
        print("Average number of random-restarts with sideways move: ", self.count_init/int(r))
        print("Average number of steps required with sideways move: ", self.steps/int(r))
        print()

#   Hill-Climbing search.
    def calc_board(self, call):
        h_dash=-1

        initial_board = state(n)
        initial_board.data = initial_board.generate_initial()
        initial_board.h = initial_board.calc_h()
        if call < 4:
            print("Search Sequence " + str(call+1) + ":")
            print("Initial state:")
            for x in initial_board.data:
                for y in x:
                    print(y, end=" ")
                print()
            print("h value:", initial_board.h)
            print()

        min_board = initial_board.data
        while h_dash !=0:
            store_min_hPos = []
            previous_board = state(n, min_board)
            previous_board.h = previous_board.calc_h()
            h_dash = previous_board.h
```

```
for i in range(int(n)):
   pos = self.find_rowPos(previous_board.data,i)
   for j in range(int(n)):
      new_pos = [None]*2
      new_pos[0]=i
      new_pos[1]=j
      if new_pos == pos:
         continue

      successor_board = state(n)
      successor_board.data = previous_board.generate_successor(pos, new_pos)
      successor_board.h = successor_board.calc_h()

      if successor_board.h <= h_dash:
         h_dash = successor_board.h
         store_pos =new_pos
         store_pos.append(successor_board.h)
         store_min_hPos.append(store_pos)

if store_min_hPos:
   l = len(store_min_hPos)-1
   while l>=0:
      y = store_min_hPos[l]
      if y[2] != h_dash:
         del store_min_hPos[l]
      l-=1

   rand = random.randint(0,len(store_min_hPos)-1)
   pos_dash = store_min_hPos[rand]
   del pos_dash[2]
   pos_parent = self.find_rowPos(previous_board.data,pos_dash[0])
   min_board = previous_board.generate_successor(pos_parent, pos_dash)

if h_dash == previous_board.h:
   if h_dash == 0:
      if call < 4:
         print("Solution found.")
      self.successes +=1
   else:
```

```
            self.tot_fSteps +=self.steps
            if call < 4:
                for x in min_board:
                    for y in x:
                        print(y, end=" ")

                    print()
                print("h value: " + str(h_dash))
                print()
                print("Solution not found.")
                print()
            self.failures +=1
        break

    else:
        self.steps +=1
        if call < 4:
            print("Next state:")
            for x in min_board:
                for y in x:
                    print(y, end=" ")

                print()
            print("h value:", h_dash)
            print()

        if h_dash==0:
            self.tot_sSteps += self.steps
            if call < 4:
                print("Solution found.")
                print()
            self.successes += 1

#   Hill-climbing search with sideways move.
    def calc_board_wSideway(self, call):
        try_ = 0
        h_dash=-1

        initial_board = state(n)
```

```
initial_board.data = initial_board.generate_initial()
initial_board.h = initial_board.calc_h()
if call < 4:
    print("Search Sequence " + str(call+1) + ":")
    print("Initial state:")
    for x in initial_board.data:
        for y in x:
            print(y, end=" ")
        print()
    print("h value:", initial_board.h)
    print()

min_board = initial_board.data
while h_dash !=0:
    store_min_hPos = []
    self.steps +=1
    check_hHigh = 0
    previous_board = state(n, min_board)
    previous_board.h = previous_board.calc_h()
    h_dash = previous_board.h
    for i in range(int(n)):
        pos = self.find_rowPos(previous_board.data,i)
        for j in range(int(n)):
            new_pos = [None]*2
            new_pos[0]=i
            new_pos[1]=j
            if new_pos == pos:
                continue

            successor_board = state(n)
            successor_board.data = previous_board.generate_successor(pos, new_pos)
            successor_board.h = successor_board.calc_h()

            if successor_board.h <= h_dash:
                h_dash = successor_board.h
                store_pos =new_pos
                check_hHigh = 1
                if successor_board.h < h_dash:
                    try_ = 0
```

```
            store_pos.append(successor_board.h)
            store_min_hPos.append(store_pos)

    if store_min_hPos:
        l = len(store_min_hPos)-1
        while l>=0:
            y = store_min_hPos[l]
            if y[2] != h_dash:
                del store_min_hPos[l]
            l-=1

        rand = random.randint(0,len(store_min_hPos)-1)
        pos_dash = store_min_hPos[rand]
        del pos_dash[2]
        pos_parent = self.find_rowPos(previous_board.data,pos_dash[0])
        min_board = previous_board.generate_successor(pos_parent, pos_dash)

    if h_dash == previous_board.h:
        if h_dash == 0:
            if call < 4:
                print("Solution found.")
                print()
            self.tot_sSteps += self.steps
            self.successes +=1
        else:
            if check_hHigh != 0:
                try_ +=1
                if call < 4:
                    print("Next state:")
                    for x in min_board:
                        for y in x:
                            print(y, end=" ")

                        print()
                    print("h value:", h_dash)
                    print()
            else:
                self.tot_fSteps +=self.steps
                self.failures +=1
```

```
                if call < 4:
                    print("Solution not found.")
                    print()
                break
            if try_ >=100:
                self.tot_fSteps +=self.steps
                self.failures +=1

                if call < 4:
                    for x in min_board:
                        for y in x:
                            print(y, end=" ")
                        print()
                    print("h value:", h_dash)
                    print()

                    print("Solution not found after 100 sideways move.")
                    print()
                break
        else:
            if call < 4:
                print("Next state:")
                for x in min_board:
                    for y in x:
                        print(y, end=" ")

                    print()
                print("h value:", h_dash)
                print()

            if h_dash==0:
                self.tot_sSteps += self.steps
                if call < 4:
                    print("Solution found.")
                    print()
                self.successes += 1

#   Random-restart.
    def calc_RandomRestart(self, sideway):
```

```
try_=0
h_dash=-1

while h_dash != 0:
    self.count_init +=1
    initial_board = state(n)
    initial_board.data = initial_board.generate_initial()
    initial_board.h = initial_board.calc_h()

    if initial_board.h == 0:
        self.successes +=1
        break

    min_board = initial_board.data
    while h_dash !=0:
        store_min_hPos = []
        check_hHigh = 0
        previous_board = state(n, min_board)
        previous_board.h = previous_board.calc_h()
        h_dash = previous_board.h
        for i in range(int(n)):
            pos = self.find_rowPos(previous_board.data,i)
            for j in range(int(n)):
                new_pos = [None]*2
                new_pos[0]=i
                new_pos[1]=j
                if new_pos == pos:
                    continue

                successor_board = state(n)
                successor_board.data = previous_board.generate_successor(pos, new_pos)
                successor_board.h = successor_board.calc_h()

                if successor_board.h <= h_dash:
                    if successor_board.h < h_dash:
                        try_ = 0
                    h_dash = successor_board.h
                    store_pos =new_pos
                    store_pos.append(successor_board.h)
```

```
            store_min_hPos.append(store_pos)
            check_hHigh = 1

if h_dash == 0 or check_hHigh == 0:
    if h_dash == 0:
        self.steps+=1
        self.tot_sSteps += self.steps
        self.successes +=1

    if check_hHigh == 0:
        self.tot_fSteps +=self.steps
        self.failures +=1
    break

if store_min_hPos:
    l = len(store_min_hPos)-1
    while l>=0:
        y = store_min_hPos[l]
        if y[2] != h_dash:
            del store_min_hPos[l]
        l-=1
    rand = random.randint(0,len(store_min_hPos)-1)
    pos_dash = store_min_hPos[rand]
    del pos_dash[2]
    pos_parent = self.find_rowPos(previous_board.data,pos_dash[0])
    min_board = previous_board.generate_successor(pos_parent, pos_dash)

if h_dash == previous_board.h:
    if sideway == 0:
        break

    if sideway == 1:
        self.steps+=1
        if check_hHigh != 0:
            try_ +=1
        if try_ >=100:
            break
else:
    self.steps+=1
```

start = board()
start.main()

Execution Results:

For 8-Queens problem:

- **Hill-Climbing search**

  Expected output:
  - Solves 14% of problem instances.
  - Takes 4 steps on average when it succeeds and 3 when it gets stuck.

  Actual output:

  For 100 runs:

```
How many Queens do you want to place on the board?
8
How many runs?
100

Hill-Climbing search:
Success Rate:  13.0 %
Failure Rate:  87.0 %
Average number of steps for successes:  3.5384615384615383
Average number of steps for failures:  3.1149425287356323
```

  For 200 runs:

```
How many Queens do you want to place on the board?
8
How many runs?
200

Hill-Climbing search:
Success Rate:  13.0 %
Failure Rate:  87.0 %
Average number of steps for successes:  4.1923076923076925
Average number of steps for failures:  3.057471264367816
```

For 300 runs:

```
How many Queens do you want to place on the board?
8
How many runs?
300

Hill-Climbing search:
Success Rate:  14.666666666666666 %
Failure Rate:  85.33333333333333 %
Average number of steps for successes:  4.090909090909091
Average number of steps for failures:  3.0
```

For 400 runs:

```
How many Queens do you want to place on the board?
8
How many runs?
400

Hill-Climbing search:
Success Rate:  15.25 %
Failure Rate:  84.75 %
Average number of steps for successes:  3.8852459016393444
Average number of steps for failures:  3.0678466076696167
```

For 500 runs:

```
How many Queens do you want to place on the board?
8
How many runs?
500

Hill-Climbing search:
Success Rate:  14.0 %
Failure Rate:  86.0 %
Average number of steps for successes:  3.9285714285714284
Average number of steps for failures:  3.0488372093023255
```

Search sequences for Hill-Climbing search:

Output for 8 queens on the board-

Search sequence 1	Search sequence 2	Search sequence 3	Search sequence 4
Initial state: _ _ _ _ _ _ Q _ _ _ _ Q _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ Q _ Q _ _ _ _ _ _ h value: 7 Next state: _ _ _ _ _ _ Q _ _ _ _ Q _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ Q _ Q _ _ _ _ _ _ h value: 3 Next state: _ _ _ _ _ _ Q _ _ _ _ Q _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ Q _ Q _ _ _ _ _ _ h value: 2 Next state: _ _ _ _ _ _ Q _ _ _ _ Q _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q	Initial state: _ _ _ _ Q _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ Q _ _ _ _ _ _ _ _ Q _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ Q h value: 9 Next state: _ _ _ _ Q _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ Q _ _ _ _ _ _ _ _ Q _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ Q _ _ h value: 6 Next state: _ _ _ _ Q _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ Q _ _ _ _ _ _ _ _ Q _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ Q _ _ h value: 4 Next state: _ _ _ _ Q _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ Q _ _ _ _ _ _ _ _ _ _ Q _	Initial state: Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ Q _ _ _ _ _ _ _ Q _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q h value: 5 Next state: Q _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ Q _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q h value: 3 Next state: _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ Q _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q h value: 1 Next state: _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ Q _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _	Initial state: _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ Q _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ h value: 7 Next state: _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ Q _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ Q _ _ _ _ h value: 4 Next state: _ _ Q _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ Q _ _ _ _ h value: 2 Next state: _ _ Q _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ Q

_ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ Q _ _ _ _ _ _ h value: 1 _ _ _ _ _ _ Q _ _ _ _ Q _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ h value: 1 Solution not found.	Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ Q _ _ h value: 1 _ _ _ _ Q _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ Q _ _ _ _ _ _ _ _ _ _ Q _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ Q _ _ h value: 1 Solution not found.	Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q h value: 1 Solution not found.	_ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ Q _ _ _ _ h value: 1 Next state: _ _ Q _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ Q _ _ _ _ _ _ _ h value: 0 Solution found.

- **Hill-Climbing search with sideways move**

  Expected output:

  - Solves 94% of problem instances.
  - Takes 21 steps on average when it succeeds and 64 on failure.

  Actual output:
  100 runs

```
How many Queens do you want to place on the board?
8
How many runs?
100

Hill-Climbing search with sideways move:
Success Rate:  94.0 %
Failure Rate:  6.0 %
Average number of steps for successes:  19.26595744680851
Average number of steps for failures:  70.16666666666667
```

200 runs

```
How many Queens do you want to place on the board?
8
How many runs?
200

Hill-Climbing search with sideways move:
Success Rate:  94.0 %
Failure Rate:  6.0 %
Average number of steps for successes:  19.50531914893617
Average number of steps for failures:  64.08333333333333
```

---

300 runs

```
How many Queens do you want to place on the board?
8
How many runs?
300

Hill-Climbing search with sideways move:
Success Rate:  95.66666666666667 %
Failure Rate:  4.333333333333333 %
Average number of steps for successes:  19.222996515679444
Average number of steps for failures:  57.84615384615385
```

---

400 runs

```
How many Queens do you want to place on the board?
8
How many runs?
400

Hill-Climbing search with sideways move:
Success Rate:  95.0 %
Failure Rate:  4.75 %
Average number of steps for successes:  19.65
Average number of steps for failures:  57.526315789473685
```

---

500 runs

```
How many Queens do you want to place on the board?
8
How many runs?
500

Hill-Climbing search with sideways move:
Success Rate:  95.0 %
Failure Rate:  5.0 %
Average number of steps for successes:  18.077894736842104
Average number of steps for failures:  61.08
```

<u>Search sequences for Hill-Climbing with sideways move implemented-</u>

Output for 8 queens on the board-

Search sequence 1	Search sequence 2	Search sequence 3	Search sequence 4
Initial state: __Q_____ ____Q___ ______Q_ __Q_____ _______Q _____Q__ ______Q_ _______Q h value: 8 Next state: __Q_____ ____Q___ ______Q_ __Q_____ _______Q _____Q__ _Q______ _______Q h value: 5 Next state: __Q_____ ____Q___ ______Q_	Initial state: ___Q____ __Q_____ ____Q___ ____Q___ ___Q____ ____Q___ ___Q____ ______Q_ h value: 13 Next state: ___Q____ __Q_____ ____Q___ ____Q___ Q_______ ____Q___ ___Q____ ______Q_ h value: 8 Next state: ___Q____ __Q_____ ____Q___	Initial state: Q_______ Q_______ Q_______ Q_______ ____Q___ Q_______ _Q______ _Q______ h value: 14 Next state: ______Q_ Q_______ Q_______ Q_______ ____Q___ Q_______ _Q______ _Q______ h value: 9 Next state: ______Q_ Q_______ Q_______	Initial state: _______Q _Q______ ______Q_ __Q_____ _Q______ _____Q__ _Q______ ____Q___ h value: 7 Next state: _______Q _Q______ ______Q_ __Q_____ Q_______ _____Q__ _Q______ ____Q___ h value: 3 Next state: _______Q _Q______ ______Q_

Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ Q h value: 3 Next state: _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ Q _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ Q h value: 2 Next state: _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ Q _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ Q _ _ h value: 2 Next state: _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ Q _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ Q _ _ h value: 1 Next state: _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ Q _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _	_ _ _ _ Q _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ Q _ h value: 4 Next state: _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ Q _ _ _ _ _ _ _ Q _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ Q _ h value: 2 Next state: _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ Q _ h value: 2 Next state: _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ Q _ h value: 1 Next state: _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _	Q _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ Q _ Q _ _ _ _ _ _ _ Q _ _ _ _ _ _ h value: 5 Next state: _ _ _ _ _ _ Q _ Q _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ Q _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ h value: 3 Next state: _ _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ Q _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ Q _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ h value: 2 Next state: _ _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ Q _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ h value: 1 Next state: _ _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q	_ _ Q _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ Q _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ h value: 2 Next state: _ _ _ _ _ _ _ Q _ _ Q _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ Q _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ h value: 2 Next state: _ _ _ _ _ _ _ Q _ _ Q _ _ _ _ _ _ _ _ _ _ _ Q _ _ Q _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ Q _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ h value: 2 Next state: _ _ _ _ _ _ _ Q _ _ Q _ _ _ _ _ _ _ _ _ _ _ Q _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ Q _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ h value: 1 Next state: _ _ Q _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ Q _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ _

<table>
<tr>
<td>_ _ _ _ _ _ _ Q
_ _ _ _ _ Q _ _
h value: 0

Solution found.</td>
<td>_ _ _ Q _ _ _ _
Q _ _ _ _ _ _ _
h value: 1

Next state:
_ Q _ _ _ _ _ _
_ _ _ _ _ _ _ Q
_ _ _ _ Q _ _ _
_ _ Q _ _ _ _ _
_ _ _ _ _ _ _ Q
_ _ _ _ _ Q _ _
_ _ _ Q _ _ _ _
Q _ _ _ _ _ _ _
h value: 1

Next state:
_ Q _ _ _ _ _ _
_ _ _ _ _ _ Q _
_ _ _ _ Q _ _ _
_ _ Q _ _ _ _ _
_ _ _ _ _ _ _ Q
_ _ _ _ _ Q _ _
_ _ _ Q _ _ _ _
Q _ _ _ _ _ _ _
h value: 1

Next state:
_ Q _ _ _ _ _ _
_ _ _ _ _ _ Q _
_ _ _ _ Q _ _ _
_ _ Q _ _ _ _ _
_ _ _ _ _ _ _ Q
_ _ _ _ _ Q _ _
_ _ _ Q _ _ _ _
_ Q _ _ _ _ _ _
h value: 1

Next state:
Q _ _ _ _ _ _ _
_ _ _ _ _ _ Q _
_ _ _ _ Q _ _ _
_ _ Q _ _ _ _ _
_ _ _ _ _ _ _ Q
_ _ _ _ _ Q _ _
_ _ _ Q _ _ _ _
_ Q _ _ _ _ _ _
h value: 1</td>
<td>_ Q _ _ _ _ _ _
_ _ _ Q _ _ _ _
h value: 0

Solution found.</td>
<td>Q _ _ _ _ _ _ _
_ _ _ _ Q _ _ _
h value: 1

Next state:
_ _ Q _ _ _ _ _
_ _ Q _ _ _ _ _
_ _ _ _ _ _ _ Q
_ Q _ _ _ _ _ _
_ _ _ Q _ _ _ _
_ _ _ _ _ Q _ _
Q _ _ _ _ _ _ _
_ _ _ _ Q _ _ _
h value: 1

Next state:
_ _ Q _ _ _ _ _
_ _ Q _ _ _ _ _
_ _ _ _ _ _ Q _
_ Q _ _ _ _ _ _
_ _ _ Q _ _ _ _
_ _ _ _ _ Q _ _
Q _ _ _ _ _ _ _
_ _ _ _ Q _ _ _
h value: 1

Next state:
_ _ Q _ _ _ _ _
_ _ Q _ _ _ _ _
_ _ _ _ _ _ _ Q
_ Q _ _ _ _ _ _
_ _ _ Q _ _ _ _
_ _ _ _ _ Q _ _
Q _ _ _ _ _ _ _
_ _ _ _ Q _ _ _
h value: 1

Next state:
_ _ Q _ _ _ _ _
_ _ _ _ Q _ _ _
_ _ _ _ _ _ _ Q
_ Q _ _ _ _ _ _
_ _ _ Q _ _ _ _
_ _ _ _ _ Q _ _
Q _ _ _ _ _ _ _
_ _ _ _ Q _ _ _
h value: 1</td>
</tr>
</table>

	Next state: _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ h value: 1 Next state: _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ h value: 1 Next state: _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ h value: 1 Next state: _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ h value: 1 Next state: _ Q _ _ _ _ _ _		Next state: _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ Q _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ h value: 1 Next state: _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ Q _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ Q _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ h value: 1 Next state: _ _ Q _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ Q _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ h value: 1 Next state: _ _ Q _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ Q _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ Q _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ h value: 1 Next state: _ _ _ _ _ _ _ Q

	_ _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ h value: 1 Next state: _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ h value: 1 Next state: _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ Q _ _ _ _ _ _ h value: 0 Solution found.		_ _ Q _ _ _ _ _ _ _ _ _ _ _ Q _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ Q _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ h value: 1 Next state: _ _ _ _ _ Q _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ Q _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ Q _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ h value: 1 Next state: _ _ _ _ _ Q _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ Q _ _ Q _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ Q Q _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ h value: 0 Solution found.

- **Random-Restart without sideways move and with sideways move**

  Expected output:

  - Without sideways move, takes roughly 7 iterations to find a goal and around 22 steps in all.
  - With sideways move, 1.06 iterations are needed on average to find a goal and around 25 steps in all.

Actual Output:

100 runs

```
How many Queens do you want to place on the board?
8
How many runs?
100

Random-Restart Hill-Climbing search without sideways move:
Average number of random-restarts without sideways move:  6.62
Average number of steps required without sideways move:  20.83

Random-Restart Hill-Climbing search with sideways move:
Average number of random-restarts with sideways move:  1.05
Average number of steps required with sideways move:  25.9
```

---

200 runs

```
How many Queens do you want to place on the board?
8
How many runs?
200

Random-Restart Hill-Climbing search without sideways move:
Average number of random-restarts without sideways move:  7.67
Average number of steps required without sideways move:  24.31

Random-Restart Hill-Climbing search with sideways move:
Average number of random-restarts with sideways move:  1.065
Average number of steps required with sideways move:  23.625
```

---

300 runs

```
How many Queens do you want to place on the board?
8
How many runs?
300

Random-Restart Hill-Climbing search without sideways move:
Average number of random-restarts without sideways move:  7.3933333333333335
Average number of steps required without sideways move:  23.663333333333334

Random-Restart Hill-Climbing search with sideways move:
Average number of random-restarts with sideways move:  1.0566666666666666
Average number of steps required with sideways move:  22.793333333333333
```

---

400 runs

```
How many Queens do you want to place on the board?
8
How many runs?
400

Random-Restart Hill-Climbing search without sideways move:
Average number of random-restarts without sideways move:  7.31
Average number of steps required without sideways move:  23.305

Random-Restart Hill-Climbing search with sideways move:
Average number of random-restarts with sideways move:  1.045
Average number of steps required with sideways move:  21.35
```

---

500 runs

```
How many Queens do you want to place on the board?
8
How many runs?
500

Random-Restart Hill-Climbing search without sideways move:
Average number of random-restarts without sideways move:  6.816
Average number of steps required without sideways move:  21.9

Random-Restart Hill-Climbing search with sideways move:
Average number of random-restarts with sideways move:  1.068
Average number of steps required with sideways move:  23.062
```

Reference:

- Artificial Intelligence -A Modern Approach, Third Edition, Stuart J. Russell and Peter Norvig
- Lecture slides