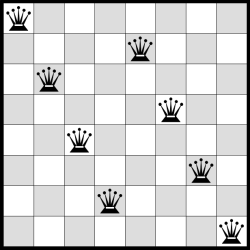
**N-queen Problem - hill climbing and its variants**



**DOCUMENTATION REPORT**

PROGRAMMING PROJECT II

**ITCS 6150 - Intelligent Systems**

DEPARTMENT OF COMPUTER SCIENCE

**SUBMITTED TO**

**Dewan T. Ahmed, Ph.D.**

**SUBMITTED BY:**

|  |  |
| --- | --- |
| **DIPESH CHAKRANI** | **ANSHU SINGH** |
| **801080983** | **801073885** |



Table of Contents

1 PROBLEM FORMULATION 2

1.1 Introduction2

1.2 Algorithm Pseudocode 3

2 PROGRAM STRUCTURE 3

2.1 Functions/Procedures 3

2.2 Global/Local variables 4

2.3 Code -------------------------------------------------------------------------------------------------------------------------5

2.4 Sample Output-----------------------------------------------------------------------------------------------------------16

3 CONCLUSIONS 18

REFERENCES 18

**PROBLEM FORMULATION**

* 1. **INTRODUCTION**

**N-queen problem:**

The eight queens puzzle is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other. Thus, no two queens should share the same row, column, or diagonal.

For example- The solution for 8-Queen problem looks something like this.

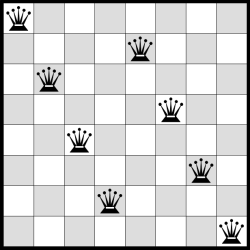


Fig 1- Solution of 8-queen problem

**Hill-climbing:**

This algorithm is simply a loop that continually moves in a direction of increasing value- uphill. It terminates when it reaches the peak where no neighbor has the highest value. The algorithm does not maintain a search tree and therefore only record the state and the value of the objective function.

**PROGRAM STRUCTURE**

**2.1 Introduction**

Number of queens are treated as N variable and input can enter the value N. Following 3 are implemented

1) Steepest-ascent hill climbing

2) Hill-Climbing with sideways move

3) Random-restart hill-climbing with and without sideways move The N-queens problem is described below by an example.

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other.

**2.2 Functions and Procedures**

* def \_\_init\_\_(self, queen\_positions=None, queen\_num=8, parent=None, path\_cost=0, f\_cost=0, side\_length=8) – it’s the path constructor for the the QueenState.
* random\_position(self) – placing each queen in a random row in a different column.
* get\_children(self) - fetching all the possible queen moves.
* random\_child(self) – selecting random child for allow sideways move algorithm
* queen\_attacks(self) – queen to check for attacking queen
* num\_queen\_attacks(self) – reporting violation
* \_\_init\_\_(self, start\_state=None) – default constructor for initial board
* goal\_test(self, state) – check if goal state is attained
* cost\_function(self, state) – calculate number of violation
* avg\_steps(result\_list, key) – calculate the average number of steps needed
* print\_data(results) – prints result of all the hill climbing algorithm
* print\_data\_row(row\_title, data\_string, data\_func, results) – print data row wise
* print\_results(results) – print results
* analyze\_performance(problem\_set, search\_function) – function takes problem set and calls passed (parameter) search function and calculates steps
* analyze\_all\_algorithms(problem\_set) – function to solve problem with steepest ascent, steepest ascent 100 sideway moves, random restart and random restart 100 sideways move
* steepest\_ascent\_hill\_climb(problem, allow\_sideways=False, max\_sideways=100) – steepest ascent hill climbing with and without sideways

**2.2 Global and Local variables**

Local variables-

* queen\_positions
* queen\_num
* parent
* path\_cost
* f\_cost
* side\_length
* parent\_queen\_positions
* random\_queen\_index
* attacking\_pairs
* start\_state
* result\_list
* results
* num\_iterations
* section\_break
* freq
* queens\_problem\_set
* children
* children\_cost
* min\_cost
* node
* node\_cost
* sideways\_moves
* path
* best\_child
* best\_child\_cost

**Program Code:**

1. **simulations.py**

from statistics import mean

from search import steepest\_ascent\_hill\_climb, random\_restart\_hill\_climb

from queens import QueensProblem

#calculates the average number of steps needed

def avg\_steps(result\_list, key):

results = [result[key] for result in result\_list]

if len(result\_list) == 1:

return {'avg': result\_list[0][key]}

elif not result\_list:

return {'avg': 0}

return {'avg': mean(results)}

#Prints results of all the hill climbing algorithms

def print\_data(results):

title\_col\_width = 30

data\_col\_width = 15

#Prints data row wise

def print\_data\_row(row\_title, data\_string, data\_func, results):

nonlocal title\_col\_width, data\_col\_width

row = (row\_title + '\t').rjust(title\_col\_width)

for result\_group in results:

row += data\_string.format(\*\*data\_func(result\_group)).ljust(data\_col\_width)

print(row)

num\_iterations = len(results[0])

#prints table headings

print('\t'.rjust(title\_col\_width) +

'All Problems'.ljust(data\_col\_width) +

'Successes'.ljust(data\_col\_width) +

'Failures'.ljust(data\_col\_width))

#print total iterations

print\_data\_row('Iterations:',

'{count:.0f}',

lambda x: {'count': len(x)},

results)

#print rates in percentages for succcess and failure

print\_data\_row('Percentage:',

'{percent:.1%}',

lambda x: {'percent': len(x) / num\_iterations},

results)

#print Average steps for success and failure

print\_data\_row('Average Steps:',

'{avg:.0f}',

lambda x: avg\_steps(x, 'path\_length'),

results)

#print (results['restarts'])

#if 'total\_nodes' in results[0][0].keys():

# print\_data\_row('Average nodes generated:',

# '{avg:.0f}',

# lambda x: avg\_steps(x, 'total\_nodes'),

# results)

#prints results

def print\_results(results):

print\_data(results)

#function takes problem set and calls passed (parameter) search function and calculates steps

def analyze\_performance(problem\_set, search\_function):

num\_iterations = len(problem\_set)

restart = 0

results = []

for problem\_num, problem in enumerate(problem\_set):

#printing 3 search sequence from 3 randon initial configurations

if problem\_num == 0 or problem\_num == 1 or problem\_num == 2:

print("Interation :" + str(problem\_num + 1))

result = search\_function(problem, problem\_num)

result['path\_length'] = len(result['solution'])-1

restart += int(result['restarts'])

results.append(result)

print(' '\*50 + '\r', end='', flush=True)

print ("Random Restart Required")

results = [results,

[result for result in results if result['outcome'] == 'success'],

[result for result in results if result['outcome'] == 'failure']]

print\_results(results)

#function to solve problem using steepest ascent, steepest ascent (100 sideways moves), random restart, randon restart (100 sideways moves)

def analyze\_all\_algorithms(problem\_set):

section\_break = '\n' + '\_'\*100 + '\n'

print(section\_break)

print('Steepest ascent hill climb (no sideways moves allowed):\n')

analyze\_performance(problem\_set, steepest\_ascent\_hill\_climb)

print(section\_break)

print('Steepest ascent hill climb (up to 100 sideways moves allowed):\n')

analyze\_performance(problem\_set, lambda x, y: steepest\_ascent\_hill\_climb(x, y, allow\_sideways=True))

print(section\_break)

print('Random restart hill climb:\n')

analyze\_performance(problem\_set, lambda x, y: random\_restart\_hill\_climb(problem\_set[0].\_\_class\_\_, y))

print(section\_break)

print('Random restart hill climb (up to 100 sideways moves allowed):\n')

analyze\_performance(problem\_set, lambda x, y: random\_restart\_hill\_climb(problem\_set[0].\_\_class\_\_, y, allow\_sideways=True))

print(section\_break)

print('N-QUEENS PROBLEMS BY HILL CLIMBING:')

#number of iterations input from user

freq=int(input("Enter Number of iterations:"))

#n=int(input('Enter Number of queens:'))

#QueensProblem to generate random queen state and calculate heuristic

queens\_problem\_set = [QueensProblem() for \_ in range(freq)]

analyze\_all\_algorithms(queens\_problem\_set)

1. **Queens.py**

from random import randrange

from copy import deepcopy

from heapq import heappop, heappush

from timeit import default\_timer as timer

from random import choice, shuffle, random

from math import exp

from search import steepest\_ascent\_hill\_climb

class QueensState:

instance\_counter = 0

#default constructor for QueensState

def \_\_init\_\_(self, queen\_positions=None, queen\_num=8, parent=None, path\_cost=0, f\_cost=0, side\_length=8):

self.side\_length = side\_length

if queen\_positions is None:

self.queen\_num = queen\_num

self.queen\_positions = frozenset(self.random\_position())

else:

self.queen\_positions = frozenset(queen\_positions)

self.queen\_num = len(self.queen\_positions)

#print ("$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$")

#print (self.queen\_positions)

self.path\_cost = 0

self.f\_cost = f\_cost

self.parent = parent

self.id = QueensState.instance\_counter

QueensState.instance\_counter += 1

#placing each queens in a random row in a different column

def random\_position(self):

open\_columns = list(range(self.side\_length))

queen\_positions = [(open\_columns.pop(randrange(len(open\_columns))), randrange(self.side\_length)) for \_ in

range(self.queen\_num)]

return queen\_positions

#fetching all the possible queen moves

def get\_children(self):

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.side\_length) if row != queen[1]]

for new\_position in new\_positions:

queen\_positions = deepcopy(parent\_queen\_positions)

queen\_positions[queen\_index] = new\_position

children.append(QueensState(queen\_positions))

return children

#selecting randon child for allow sideways move algorithm

def random\_child(self):

queen\_positions = list(self.queen\_positions)

random\_queen\_index = randrange(len(self.queen\_positions))

queen\_positions[random\_queen\_index] = (queen\_positions[random\_queen\_index][0],

choice([row for row in range(self.side\_length) if row != queen\_positions[random\_queen\_index][1]]))

return QueensState(queen\_positions)

#function to check for attacking queens

def queen\_attacks(self):

def range\_between(a, b):

if a > b:

return range(a-1, b, -1)

elif a < b:

return range(a+1, b)

else:

return [a]

def zip\_repeat(a, b):

if len(a) == 1:

a = a\*len(b)

elif len(b) == 1:

b = b\*len(a)

return zip(a, b)

def points\_between(a, b):

return zip\_repeat(list(range\_between(a[0], b[0])), list(range\_between(a[1], b[1])))

def is\_attacking(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 points\_between(a, b):

if between in queens:

return False

return True

else:

return False

attacking\_pairs = []

queen\_positions = list(self.queen\_positions)

left\_to\_check = deepcopy(queen\_positions)

while left\_to\_check:

a = left\_to\_check.pop()

for b in left\_to\_check:

if is\_attacking(queen\_positions, a, b):

attacking\_pairs.append([a, b])

return attacking\_pairs

#reporting violations

def num\_queen\_attacks(self):

return len(self.queen\_attacks())

def \_\_str\_\_(self):

return '\n'.join([' '.join(['.' if (col, row) not in self.queen\_positions else '\*' for col in range(

self.side\_length)]) for row in range(self.side\_length)])

def \_\_hash\_\_(self):

return hash(self.queen\_positions)

def \_\_eq\_\_(self, other):

return self.queen\_positions == other.queen\_positions

def \_\_lt\_\_(self, other):

return self.f\_cost < other.f\_cost or (self.f\_cost == other.f\_cost and self.id > other.id)

class QueensProblem:

#default constructor for initail board

def \_\_init\_\_(self, start\_state=None):

if not start\_state:

start\_state = QueensState()

self.start\_state = start\_state

#check if goal sate is attained

def goal\_test(self, state):

return state.num\_queen\_attacks() == 0

#calculate number of violations

def cost\_function(self, state):

return state.num\_queen\_attacks()

1. **search.py**

from random import choice, random

from math import exp

from heapq import heappop, heappush

#steepest ascent hill climbing with and without sideways moves

def steepest\_ascent\_hill\_climb(problem, problem\_num, allow\_sideways=False, max\_sideways=100):

#funtion to get next best state (queen move)

def get\_best\_child(node, problem):

children = node.get\_children()

children\_cost = [problem.cost\_function(child) for child in children]

min\_cost = min(children\_cost)

best\_child = choice([child for child\_index, child in enumerate(children) if children\_cost[

child\_index] == min\_cost])

return best\_child

node = problem.start\_state

node\_cost = problem.cost\_function(node)

path = []

sideways\_moves = 0

while True:

#print 3 search sequence from 3 randon initial configurations

#uncomment to print path

if problem\_num == 0 or problem\_num == 1 or problem\_num == 2:

print (node)

print ('\n')

path.append(node)

best\_child = get\_best\_child(node, problem)

best\_child\_cost = problem.cost\_function(best\_child)

if best\_child\_cost > node\_cost:

break

elif best\_child\_cost == node\_cost:

if not allow\_sideways or sideways\_moves == max\_sideways:

break

else:

sideways\_moves += 1

else:

sideways\_moves = 0

node = best\_child

node\_cost = best\_child\_cost

return {'outcome': 'success' if problem.goal\_test(node) else 'failure',

'solution': path,

'problem': problem,

'restarts': 0}

#random restart hill climbing with and without sideways moves

def random\_restart\_hill\_climb(random\_problem\_generator, problem\_num, num\_restarts=100, allow\_sideways=False, max\_sideways=100):

path = []

restarts = 0

for \_ in range(num\_restarts):

result = steepest\_ascent\_hill\_climb(random\_problem\_generator(), problem\_num, allow\_sideways=allow\_sideways,

max\_sideways=max\_sideways)

path += result['solution']

#counter to count randon restart

if result['outcome'] == 'failure':

restarts += 1

if result['outcome'] == 'success':

break

result['solution'] = path

result['restarts'] = restarts

return result

**Sample Output:**

1. For 100 iteration, please see the file “Output for 100 iteration” in this folder.
2. For 200 iteration, please see the file “Output for 200 iteration” in this folder.
3. For 300 iteration, please see the file “Output for 300 iteration” in this folder.
4. For 400 iteration, please see the file “Output for 400 iteration” in this folder.
5. For 500 iteration, please see the file “Output for 500 iteration” in this folder.
   1. **References**

* <https://en.wikipedia.org/wiki/Hill_climbing>
* <https://en.wikipedia.org/wiki/Eight_queens_puzzle>