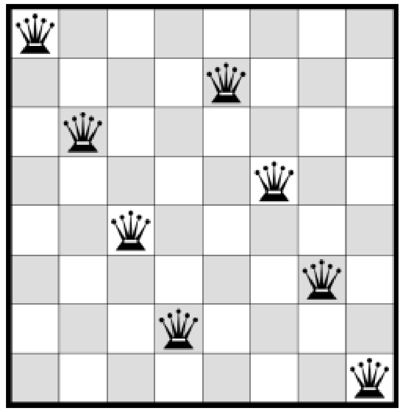
N-queen Problem - hill climbing and its variants



DOCUMENTATION REPORT

PROGRAMMING PROJECT II

ITCS 6150 - Intelligent Systems

DEPARTMENT OF COMPUTER SCIENCE

SUBMITTED TO
Dewan T. Ahmed, Ph.D.

SUBMITTED BY:

DIPESH CHAKRANI

801080983

ANSHU SINGH

801073885

UNC CHARLOTTE
College of Computing and Informatics

Table of Contents

PROBLEM FORMULATION

1.1 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.

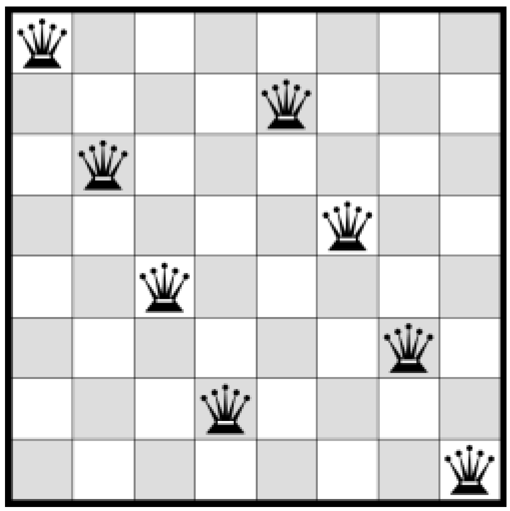


Fig 1- Solution of 8-queen problem

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('\n')
    print ("Average Random Restart Required:"+ str((restart + freq) / int(freq)) + '\n')

    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)
```

2. 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()
```

3. 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.

3.1 References

- https://en.wikipedia.org/wiki/Hill_climbing
- https://en.wikipedia.org/wiki/Eight_queens_puzzle