Assignment 1

AI

By:

Sultan Aljohani (391002603)

Omer Khan (391007603)

Supervised By:

Dr. Abdullah Al-Shanqiti

الجامعة الإسلامية بالمدينة المنورة

ISLAMIC UNIVERSITY OF MADINAH

Faculty of Computer and Information Systems

Islamic University of Madinah

Contents

List of Figures

1 Task 1: cops-robbers problem

1.1 Introduction

The cops-robbers problem is game that consist of cops and robbers in same side of mountain and they have to move to other side of the river using a cable car , from reading the description given in the assignment we have to follow following rules:

- the number of robbers on any side cannot out number the number of cops
- only one or two person can be in the cable car at the same time

after reading the problem description there are some unclear points such as

1. can robbers stay alone in one side?
2. can the robber function the cable car
3. if two persons are in the cable do they have to step out then doing next move or one person can stay for example :
   2 robbers in cable car going to side where there is only 1 cop so in this situation is it possible for one robber to go back without leaving the cable car or it will consider not acceptable situation (2 robbers > 1 cop)

to decide on which rules we will work , we review some related problems such as sheep's and wolves and missionaries and cannibals where we find in these problems rules that it's acceptable for robbers to stay alone , and staying in the cable car is also acceptable , therefore we analyse the solution and we find that its impossible to take all the three points in consideration but there is solution where robbers cannot be alone which make more sense logically

1.2 State representation

upSide and downSide represent number of cops and robbers in each side , C represent one cop , R represent one robber , CC represent the cable car, for example :
this one state where 1 cop and 1 robber move to another side
upSide(C,C,R,R) , downSide(C,R,CC)

1.3 initial and goal states

initial stat : 3 cops 3 robbers on the up side
actions :

- 1 cop move
- 1 robber move
- 2 cops move
- 2 robbers move
- 1 cop 1 robber move

goal stat : 3 cops 3 robbers on the down side

1.4 successor function

considering cops and/or robbers that move together as set (it will be called Movers) can be moved from up side to down side if :

- The cable car is on there side.
- The set people move must consists of 1 or 2 people that are on up side.
- The number of cops in up side is set by subtracting Movers from up side it must be 0 or greater than or equal to the number of robbers.
- The number of cops in down side is set by adding Movers to down side it must be 0 or greater than or equal to the number of robbers.

1.5 cost function

each move will be consider as cost , therefore for each acceptable solution number of moves will be stored then the solution with lowest cost will be considered as optimal solution

2 Task 2: N-Queens Problem - Simulated Annealing

2.1 Introduction

The N queens problem is placing N chess queens on an N x N chessboard where each queen can move vertically, horizontally, or diagonally , with restriction of that no two queens pose a threat to each other.Simulated annealing algorithm provides a stochastic optimization technique to efficiently explore large search spaces and find satisfactory solutions to this problem. The algorithm generates a random solution and incrementally improves it by examining nearby solutions using random disruption and acceptance criteria. The cooling schedule is used to control the trade-off between exploration and exploitation, allowing the algorithm to leave the local optimum and reach the global optimum. Therefore solving the N-Queens problem using simulated annealing is a general and effective approach to solve the problem and it has proven effective in solving the N-Queens problem for large N values where traditional brute force methods become impractical.

2.2 code

```python
import random
import math

def generateStartStat(n):
    # this function will genrate starting postion randomly depending on n value
    return [random.randint(0, n-1) for i in range(n)]

def computeConflicts(state):
    # this function will find conflit
    n = len(state) # size of problem
    conflicts = 0
    for i in range(n):
        for j in range(i+1, n):
            if (state[i] == state[j]) or (abs(state[i] - state[j]) == j - i):
                # first if condition to find vertically, horizontally conflict
                # second if condition to find diagonally conflict
                conflicts += 1
    return conflicts

# function to make next move
def move(state):
    n = len(state)
    i = random.randint(0, n-1)
    j = random.randint(0, n-1)
    new_state = state[:]
    new_state[i] = j
    # next state is genrated randomly
    return new_state

# function to compute probability of acceptence for each state
# probability of acceptence i computed based on -MetropolisHastings algorithm
def acceptanceProbability(delta, temperature):
```

```
    if delta < 0:
        return 1.0
    else:
        return math.exp(-delta / temperature)

# function to solve the problem using simulated annealing
def simulatedAnnealing(n, max_iter, initial_temperature, cooling_rate):
    # first state generated randomly
    current_state = generateStartStat(n)
    # energy is number of conflicts in stat
    current_energy = computeConflicts(current_state)
    # start tempature defined statcily
    temperature = initial_temperature
    for i in range(max_iter):
        # if there is no conflict
        if current_energy == 0:
            # sloution is found
            return current_state
        # generate new state
        new_state = move(current_state)
        # compute conflict for new stat
        new_energy = computeConflicts(new_state)
        # delte is diffrent of conflict between old state and current
        delta = new_energy - current_energy
        # calclaute ap of current stat
        ap = acceptanceProbability(delta, temperature)
        # decide if new state is acceptable depending on MetropolisHastings algorithm
        if random.random() < ap:
            current_state, current_energy = new_state, new_energy
        temperature *= cooling_rate
    # if sloution doesnt find under 1000 loops
    return None

def displayBoard(array=[]):
    # function display queen in board
    board = []
    for row in range(len(array)):
        # create lines in number N
        line = ""
        for col in range(len(array)):
            if array[row] == col:
                # for each line
                # if its place of queen
                line += " ||"
            else:
                line += "| |"
        board.append(line)
    return board

N = int(input("enter number of queens \n"))
solution = simulatedAnnealing(n=N, max_iter=1000, initial_temperature=1000, cooling_rate=0.95)
if solution:
    # if there is solution
    print("\nsolution as Row & postion")
    print("Row","Pos")
    for i in solution:
```

```python
        print("{} , {}".format(solution.index(i),i))
    print("\nsolution as board\n")
    b = displayBoard(solution)
    for line in b:
        print(line)
else:
    # if there is no solution
    print("No solution found")
```

Listing 1: python code for N-Queens Problem - Simulated Annealing

2.3 performance comparing

Depth-First as blind search algorithm it will explores all possible paths from the initial state to the goal state, and Simulated Annealing is a probabilistic algorithm that can escape local optima and explore a wider range of the search space. therefore in term of performance DFS can be good if N size is small but for big N size DFS maybe will take a lot of time and resources to find the solution or even it will fail , in other hand Simulated Annealing performance maybe will not be best in small N size problems comparing to DFS but in big N size problems it will be better than DFS.

In summury DFS is more easy to implement than Simulated Annealing and it perform well in small N size problems , Simulated Annealing is harder to implement but it perform well in big N size problems

3 Task 3: N-Queens Problem – Genetic Algorithm

3.1 Introduction

The N queens problem is placing N chess queens on an N x N chessboard where each queen can move vertically, horizontally, or diagonally , with restriction of that no two queens pose a threat to each other.A genetic algorithm is a type of evolutionary algorithm that uses the principle of natural selection to find solutions to optimization problems. It works by encoding a population of candidate solutions as strings of genetic information and using crossover and mutation operators to generate new solutions. This approach is very effective in finding good solutions to the N queens problem, especially for large values of N where classic methods may not be feasible.

3.2 code

```python
import random

N = int(input("enter number of queens \n"))

# Population size
popSize = 100

# Mutation probability
mutProb = 0.1

# Fitness function
def fit(board):
    attacks = 0
    for i in range(N):
        for j in range(i+1, N):
            if board[i] == board[j] or board[i] - i == board[j] - j or \
               board[i] + i == board[j] + j:
                attacks += 1
    return N - attacks

# Generate initial population
def genPop():
    pop = []
    for i in range(popSize):
        indi = list(range(N))
        random.shuffle(indi)
        pop.append(indi)
    return pop

# Select parents for crossover
def selcPop(pop):
    fitns = [fit(board) for board in pop]
    # total fitness
    totFit = sum(fitns)
    # probabilities
    porb = [f/totFit for f in fitns]
    parent1 = random.choices(pop, weights=porb)[0]
    parent2 = random.choices(pop, weights=porb)[0]
    return parent1, parent2

```

```python
# Perform crossover on parents
def crossover(parent1, parent2):
    crossover_point = random.randint(1, N-1)
    child1 = parent1[:crossover_point] + parent2[crossover_point:]
    child2 = parent2[:crossover_point] + parent1[crossover_point:]
    return child1, child2

# Mutate an individual
def mutate(indi):
    if random.random() < mutProb:
        i, j = random.sample(range(N), 2)
        indi[i], indi[j] = indi[j], indi[i]

# Run the genetic algorithm
def geneticAlgorithm():
    pop = genPop()
    for i in range(1000):
        # Select parents and perform crossover
        parent1, parent2 = selcPop(pop)
        child1, child2 = crossover(parent1, parent2)

        # Mutate the children
        mutate(child1)
        mutate(child2)

        # Add the children to the population
        pop.extend([child1, child2])

        # Remove the worst individuals from the population
        pop = sorted(pop, key=fit, reverse=True)[:popSize]

        # Check if a solution has been found
        best_fit = fit(pop[0])
        if best_fit == N:
            return pop[0]

    return None

def displayBoard(array=[]):
    # function display queen in board
    board = []
    for row in range(len(array)):
        # create lines in number N
        line = ""
        for col in range(len(array)):
            if array[row] == col:
                # for each line
                # if its place of queen
                line += " ||"
            else:
                line += "| |"
        board.append(line)
    return board


solution = geneticAlgorithm()
if solution:
    print("\nsolution as Row & postion\n")
```

```
    print("Row","Pos")
    for i in solution:
        print("{} , {}".format(solution.index(i),i))
    print("\nsolution as board\n")
    b = displayBoard(solution)
    for line in b:
        print(line)
else:
    print("No solution found")
```

Listing 2: python code for N-Queens Problem - Genetic Algorithm

3.3 performance comparing

we use time library to measure performance of each algorithm what we found out is that Simulated Annealing is slower than Genetic Algorithm in small N size problems but in big N size problems the Genetic Algorithm fails even to find the solution in less than 1000 loop while Simulated Annealing find the solution as show blow:

Figure 1: different between Simulated Annealing and Genetic Algorithm for N = 4

as we can see execution time for Simulated Annealing is 2.20 and for Genetic Algorithm is 1.45 so Simulated Annealing is slower by 0.75 second for N = 4

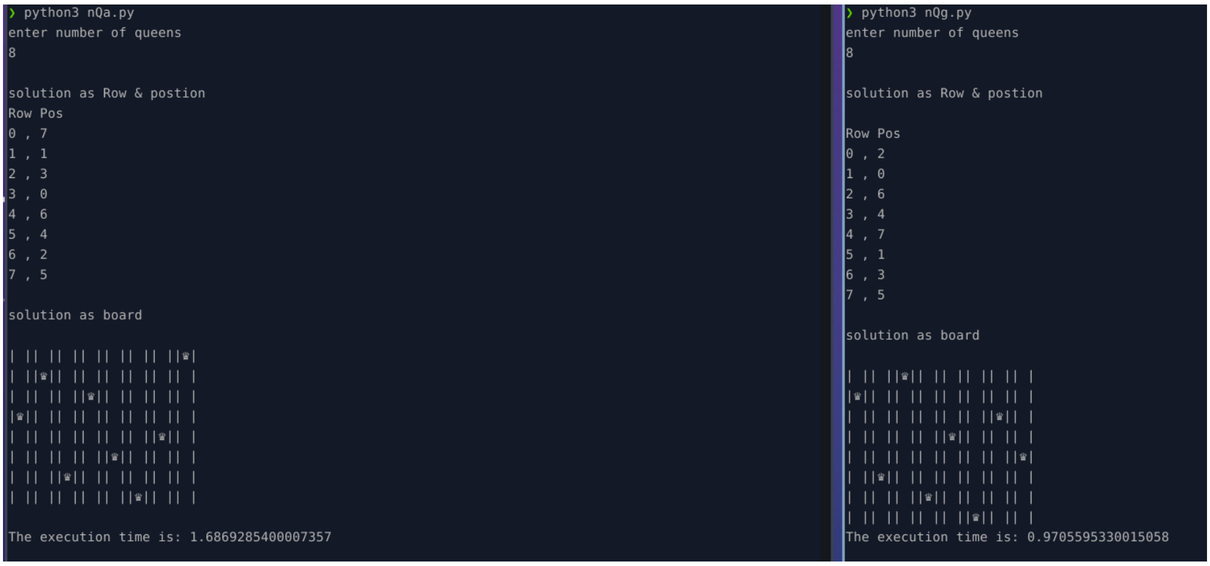

Figure 2: different between Simulated Annealing and Genetic Algorithm for N = 8

in this example N = 8 which is max number that Genetic Algorithm can find solution for it in less than 1000 loop from our experiment
as we can see execution time for Simulated Annealing is 1.69 and for Genetic Algorithm is 0.97 so Simulated Annealing is slower by 0.72 second for N = 8

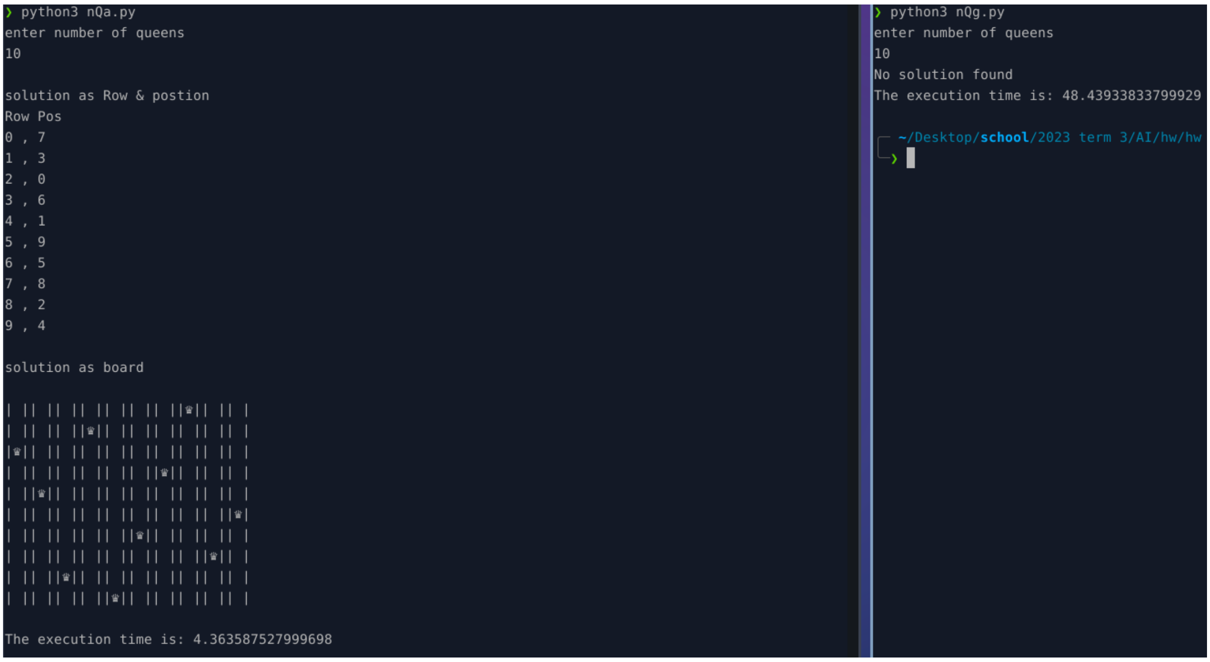

Figure 3: different between Simulated Annealing and Genetic Algorithm for N = 10

as we can see execution time for Simulated Annealing is 4.36 and for Genetic Algorithm it failed to find the solution in less than 1000 loop for N = 10