N-Queen Problem

November 10, 2019

GAA Project 2019. Course teacher: ***Dr. Vibhor Kant***

Team Member	Roll Number
Ansh Mittal	17UCS028
Anshul Jain	17UCS029
Anshul Kiyawat	17UCS030
Anshu Musaddi	17UCS185

1 Importing the required libraries

[13]:
```
import random
import pandas as pd
import numpy as np
```

2 Defining the Problem

N-Queen problem compells to find a solution of arranging N Queens on an N x N ChessBoard in such a way that it is impossible for any queen to attack each other. For example: The shown arrangement is a solution for N-Queen problem with n = 4.

0	1	2	3	4
1	x	Q	x	x
2	x	x	x	Q
3	Q	x	x	x
4	x	x	Q	x

3 Defining the functions required for Genetic Algorithm

3.1 Fitness Function

We have defined the fitness function to calculate the number of possible diagonal collisions for a chromosome. Where each chromosome is defined as 1D array(arr) containing representation of whole N x N chessboard with queens placed at row arr[i] and column i where index i ranges from [1,N].

[14]:
```
def fitness(chrm):
    cnt = 0
    for i in range(len(chrm)):
        for j in range(len(chrm)):
            if i==j:
                continue
            if abs(i-j)==abs(chrm[i]-chrm[j]):
                cnt = cnt+1
    return cnt
```

3.2 Population Initialization

The following function initializes a unique random chromosomes. The population size will be of size n passed as a parameter. The following function will return a Pandas dataframe consisting of Chromosomes and their respective fitness values

[15]:
```
def randinit(n,lenchrm):
    df = pd.DataFrame(np.zeros((n,2)),columns=["Chromosome","Fitness"],␣
↪dtype=object)
    i = 0
    arr = list(range(lenchrm))
    while i < n:
        random.shuffle(arr)
        if arr not in list(df["Chromosome"]):
            df.at[i,"Chromosome"] = arr.copy()
            df.at[i,"Fitness"] = fitness(arr)
            i = i+1
    df["Fitness"] = df["Fitness"].astype('int64')
    return df
```

3.3 Crossover

We are using **cut and crossfill** crossover where the crossover point is in the middle of chromosome.

[16]:
```
def crossover(chrm1,chrm2):
    n = len(chrm1)
    chrm3 = chrm1[:int(n/2)]
    for i in range(int(n/2), int(n/2)+n):
        if chrm2[i%n] not in chrm3:
            chrm3.append(chrm2[i%n])
    n = len(chrm2)
    chrm4 = chrm2[:int(n/2)]
    for i in range(int(n/2), int(n/2)+n):
        if chrm1[i%n] not in chrm4:
            chrm4.append(chrm1[i%n])

    return chrm3, chrm4
```

3.4 Mutation

We are performing **random swapping** between any two allele in the chromosome. It is ensured that swapping points are different.

[17]:
```
def mutation(chrm):
    if random.random() < 1/len(chrm):
        n = len(chrm)
        a = random.randint(0, n-1)
        b = random.randint(0, n-1)
        while b==a:
            b = random.randint(0, n-1)
        chrm[a], chrm[b] = chrm[b], chrm[a]
    return chrm
```

3.5 Parent Selection

For parent selection **best 2 out of random 5** chromosomes are chosen with repetition **not** allowed.

[18]:
```
def giveParent(pop):
    n = pop.shape[0]
    parents = pd.DataFrame(np.zeros((5,2)),columns=['Parents',"Fitness"],␣
 ↪dtype=object)
    i = 0
    while len(parents) <= 5:
        r = random.randint(0, n-1)
        parent = pop.iloc[r, 0]
        fitness = pop.iloc[r,1]
        if parent not in list(parents["Parents"]):
            parents.at[i,"Parents"] = parent
            parents.at[i,"Fitness"] = fitness
            i = i+1

    parents = parents.sort_values(by=['Fitness'])
    return parents.iloc[0, 0], parents.iloc[1, 0]
```

3.6 Survival Selection

The bottom 2 worst chromosomes are replaced with the new better chromosomes. This is also called **Replace worst**.

[19]:
```
def replaceWorst(p, v=False):
    pop = p.copy()
    p1, p2 = giveParent(pop)
    c1, c2 = crossover(p1, p2)
    c1 = mutation(c1)
    c2 = mutation(c2)
    if v:
```

```
        print('Child 1:{0}\tFitness: {1}\nChild 2:{2}\tFitness: {3}'.format(c1,␣
↪fitness(c1), c2, fitness(c2)))
    pop = pop.sort_values(by=['Fitness'])
    pop = pop.reset_index(drop=True)
    pop.loc[p.shape[0]-2] = [c1, fitness(c1)]
    pop.loc[p.shape[0]-1] = [c2, fitness(c2)]
    return pop
```

4 Utility Functions

The following function ensures that weather the population has the Fittest chromosome.

[20]:
```
def hasSolution(p):
    for value in p['Fitness']:
        if value==0:
            return True
    return False
```

The following function finds and returns the fittest possible chromosome from the given population.

[21]:
```
def getSolution(p):
    n = 0
    for value in p['Fitness']:
        if value==0:
            return p.iloc[n, 0]
        n += 1
```

The following function gives the fitness of the fittest chromosome from the population in return.

[22]:
```
def bestFitness(pop):
    p = pop.sort_values(by=['Fitness'])
    return p.iloc[0, 1]
```

The following function gives the fittest chromosome from the population in return.

[49]:
```
def bestChromosome(pop):
    p = pop.sort_values(by=['Fitness'])
    return p.iloc[0, 0].copy()
```

Generates a 2D array of Chess board with queens represented as Q and blanks are denoted by x

[68]:
```
def placeQueens(sol):
    board = np.zeros((len(sol), len(sol)))
    for i, j in zip(sol, range(len(sol))):
        board[i][j] = 1
    return board
```

Prints the Chess Board returned by placeQueens function.

[69]:
```
def showBoard(board):
    for i in range(np.shape(board)[0]):
        for j in range(np.shape(board)[1]):
            if board[i][j] == 0:
                print("x ", end='')
            else:
                print('Q ', end='')
        print('\n')
```

5 Execution of the Genetic Algorithm

This program accepts user chosen population size and number of queens although for representation purposes report is shown on values 200 and 8 respectively for above variables. Moreover a total of 10,000 generations is set as upper limit on number of generations.

[62]:
```
%%time

N = int(input('Value of N: '))
pop_size = int(input('Population Size: '))
show_progress = False
lastgeneration = 0

generationInfo = pd.DataFrame(np.zeros((10000, 2)), columns=['Chromosome',␣
 ↪'Fitness'], dtype=object)

for generation in range(10000):
    lastgeneration = generation
    if generation == 0:
        pop = randinit(pop_size, N)

    generationInfo.loc[generation] = [bestChromosome(pop), bestFitness(pop)]

    if hasSolution(pop):
        print("Found Solution. {:,} Generations generated.".format(generation))
        break

    if(show_progress):
        print("\nGeneration {}".format(generation))

    pop = replaceWorst(pop, show_progress)

generationInfo = generationInfo[generationInfo['Chromosome']!=0]

if(not hasSolution(pop)):
    print("Solution not found program ended.")
```

```
Value of N: 8
Population Size: 200
Found Solution. 7 Generations generated.
CPU times: user 86.9 ms, sys: 0 ns, total: 86.9 ms
Wall time: 2.94 s
```

5.1 Generated Solution

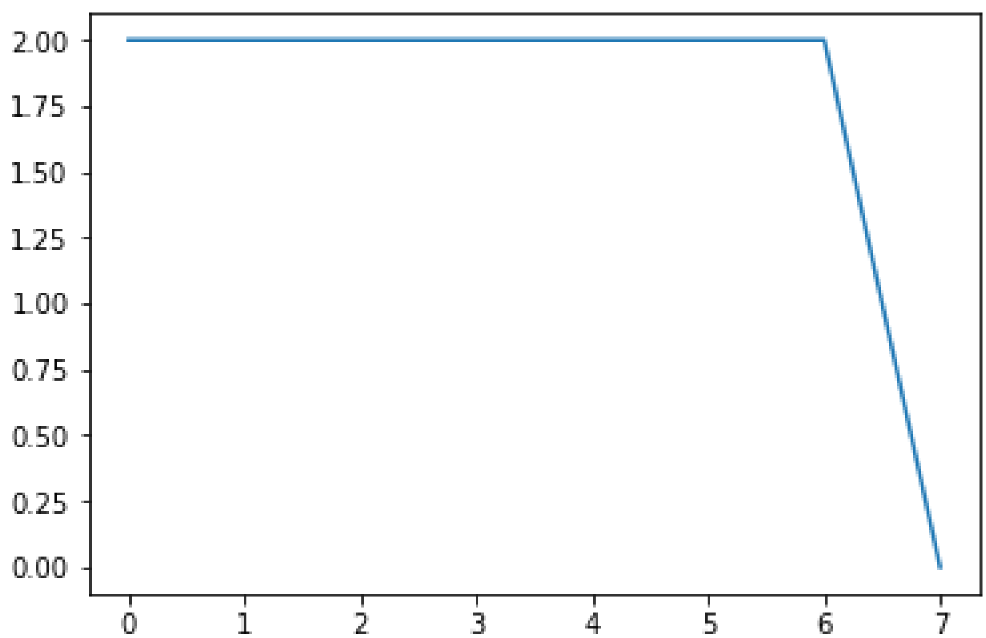
[63]:
```python
if(hasSolution(pop)):
    print(getSolution(pop))
else:
    print("No solution found.")
```

```
[6, 4, 2, 0, 5, 7, 1, 3]
```

6 Performance of the Genetic Algorithm For Each Generation

Plot of the best fitness in a generation versus generation number.

[66]:
```python
import matplotlib.pyplot as plt
plt.plot(range(generationInfo.shape[0]), list(generationInfo['Fitness']))
plt.show()
```



7 Solution on Chess Board

[70]:
```
showBoard(placeQueens(s))
```

```
x x x x Q x x x

x Q x x x x x x

x x x Q x x x x

x x x x x Q x x

x x x x x x x Q

x x Q x x x x x

Q x x x x x x x

x x x x x x Q x
```