Submitted by Harsh Srivastava

Roll - 117CS0755

importing libraries

In [1]:

```
import random
```

fitness function

depends upon the number of collision in the horizontal and diagonal directions

maximum fitness in case of no collissions

maximum fitness = (n * (n - 1)) / 2 ,where n --> number of queens

In [2]:

```
def fitness(c):
    h_cols = sum([c.count(queen)-1 for queen in c])/2
    d_cols = 0

    n = len(c)
    left_diagonal = [0] * 2*n
    right_diagonal = [0] * 2*n
    for i in range(n):
        left_diagonal[i + c[i] - 1] += 1
        right_diagonal[len(c) - i + c[i] - 2] += 1

    d_cols = 0
    for i in range(2*n-1):
        counter = 0
        if left_diagonal[i] > 1:
            counter += left_diagonal[i]-1
        if right_diagonal[i] > 1:
            counter += right_diagonal[i]-1
        d_cols += counter / (n-abs(i-n+1))

    return int(maxFitness - (h_cols + d_cols))
```

function for creating random chromosomes with binary values

In [3]:

```
def random_chromosome(size): #making random chromosomes
    return [ random.randint(1, size) for _ in range(size) ]
```

function for probability calculation

found by dividing fitness by the maximum fitness

so that all values are between 0 and 1

In [4]:

```
def probability(c, fitness):
    return fitness(c) / maxFitness
```

picking chromosomes for cross over

In [5]:

```
def random_pick(population, probabilities):
    popProbabilty = zip(population, probabilities)
    total = sum(w for c, w in popProbabilty)
    r = random.uniform(0, total)
    upto = 0
    for c, w in zip(population, probabilities):
        if upto + w >= r:
            return c
        upto += w
    assert False
```

doing cross_over between two chromosomes

In [6]:

```
def reproduce(x, y):
    c = random.randint(0, len(x) - 1)
    return x[0:c] + y[c:len(x)]
```

function for mutation

in this case we randomly chnage the bit value in the chromosome

has a very low probability

In [7]:

```
def mutate(x):
    c = random.randint(0, len(x) - 1)
    m = random.randint(1, len(x))
    x[c] = m
    return x
```

GA algo implementation

In [8]:

```
def queen_da(population, fitness, maxFitness):
    pm = 0.03
    new_population = []
    max_fit_current = -1
    max_fit_chrom = None
    probabilities = [probability(n, fitness) for n in population]
    for i in range(len(population)):
        x = random_pick(population, probabilities)
        y = random_pick(population, probabilities)
        child = reproduce(x, y)
        if random.random() < pm:
            child = mutate(child)
        fit_child = fitness(child)
        if fit_child > max_fit_current:
            max_fit_chrom = child
            max_fit_current = fit_child
        new_population.append(child)
        if fitness(child) == maxFitness: break
    print_chromosome(max_fit_chrom)
    return new_population
```

printing a chromosome value

In [9]:

```
def print_chromosome(chrom):
    print("Chromosome = {},  Fitness = {}"
        .format(str(chrom), fitness(chrom)))
```

taking number of queens

In [10]:

```
num_queens = 4
```

code block for number of queens input and simulation

In [11]:

```
maxFitness = (num_queens*(num_queens-1))/2
population = [random_chromosome(num_queens) for _ in range(100)]

generation = 1

while not maxFitness in [fitness(chrom) for chrom in population]:
    print("For Generation {}".format(generation))
    population = queen_da(population, fitness, maxFitness)
    print()
    print("Maximum Fitness = {}".format(max([fitness(n) for n in population])))
    generation += 1
chrom_out = []
print("Generations needed = {}".format(generation-1))
for chrom in population:
    if fitness(chrom) == maxFitness:
        print()
        print("Possible Solution: ")
        chrom_out = chrom
        print_chromosome(chrom)
```

```
For Generation 1
Chromosome = [2, 4, 1, 3],  Fitness = 6

Maximum Fitness = 6
Generations needed = 1

Possible Solution:
Chromosome = [2, 4, 1, 3],  Fitness = 6
```

In [ ]: