Analysing the Convergence Rate of Genetic Algorithms

CS-3111

Dr Sudip Sanyal

Group Members

Arjun Bakshi - BT18GEC134

Somanshu Singh - BT18GCS111

Yukta Sharma - BT18GCS181

The Basic Problem

Most Dominating Set of Queens Problem

Given a squared chess board we have to find the position of queens such that they dominate/threaten the maximum squares of the board.

$$A = [00000110 \quad 00010000]$$

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Genetic Algorithm

- An approach towards solving a problem by considering a large population of possible solutions and then applying the theory of evolution on the population.
- This yields fitter and fitter generations.

MDSQP

- Analogous to a popular classical puzzle game called the n-queens problem.
- With the use of this problem, we aim to test out various genetic operators and analyze their results.

Methodology

Expressed as an equation:

$$E\big(m(H,t+1)\big) \geq \frac{m(H,t)f(H)}{a_t}[1-p]$$

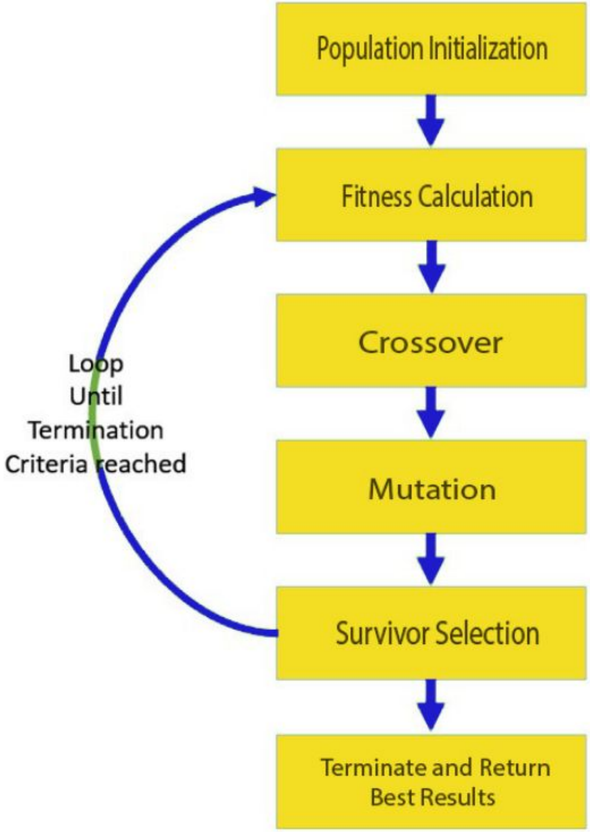
Fitness Calculation - Calculating the fitness of each chromosome in the population.

Selection - Selecting the best/strongest chromosomes in the population for crossover.

CrossOver : Selecting the best features of the parent to be added in the offspring.

Mutation -Ensuring the same population is not passed on. Deliberately Inserting some variations in some individuals of the population

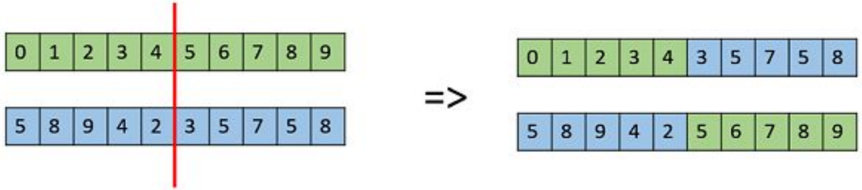
Survivor Selection: Removing the weakest of the population as each generation comes along.

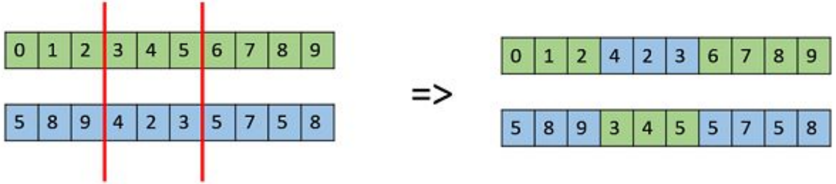


Methodology(Contd)

Types of Crossovers:

1. Single Point Crossover
2. Two point Crossover
3. Uniform Crossover



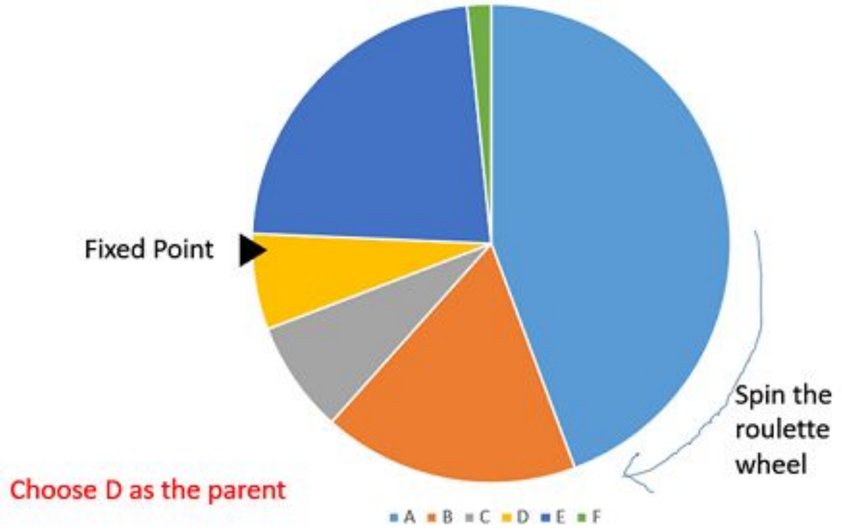


Methodology(Contd)

Types of Selections:

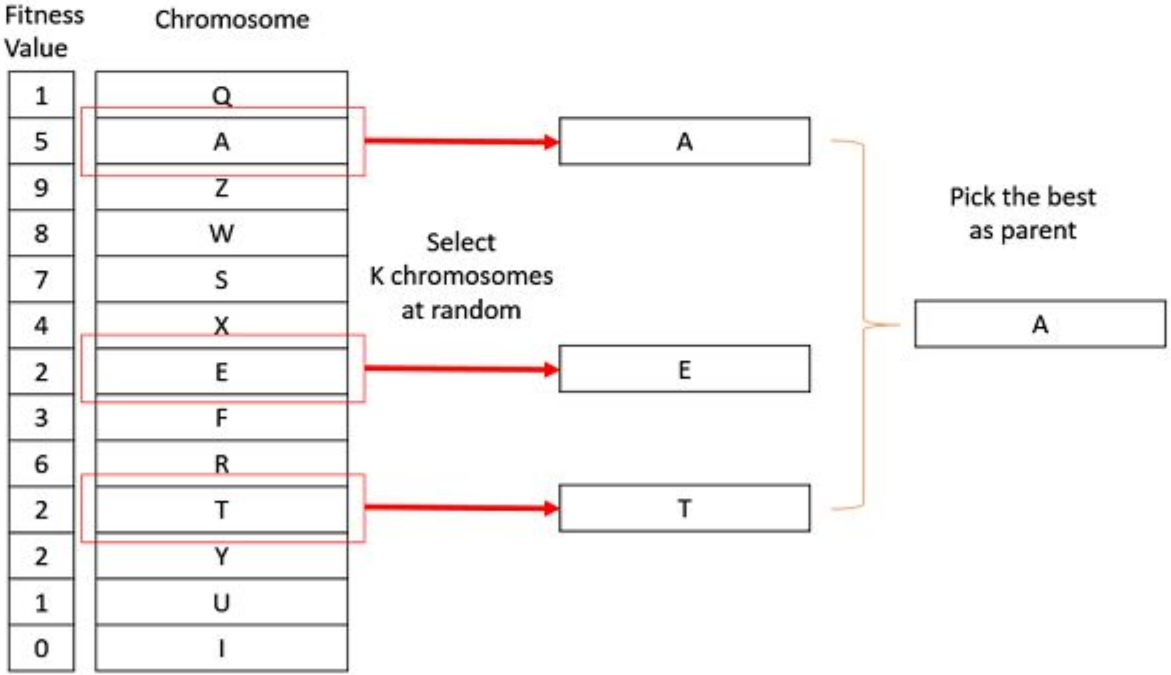
1. Elitist selection
2. Roulette-wheel selection
3. Tournament selection
4. Rank Based Selection

Methodology(Contd)

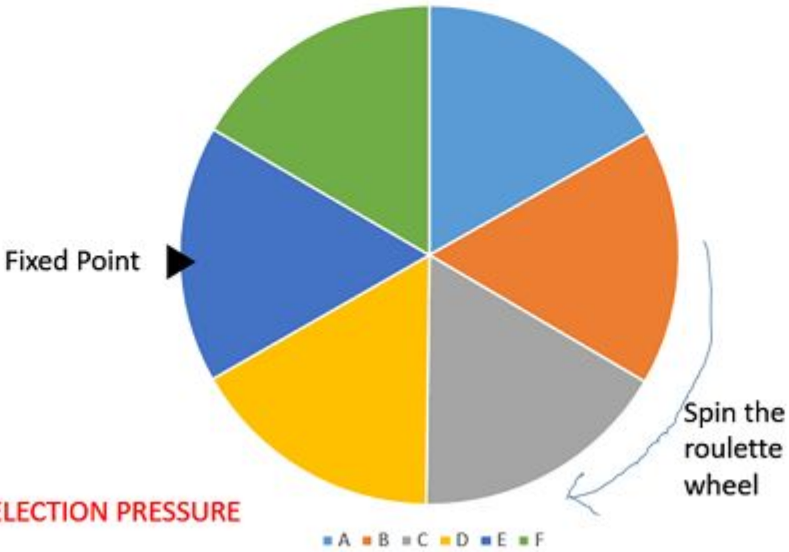


Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

Methodology(Contd)



Methodology(Contd)



NO SELECTION PRESSURE

Chromosome	Fitness Value
A	8.1
B	8.0
C	8.05
D	7.95
E	8.02
F	7.99

Methodology(Contd)

Types of Mutation:

1. Bit Flip Mutation
2. Swap Mutation
3. Scramble Mutation

Libraries

1. **GeneAL**- for genetic algorithm implementation

2. **Matplotlib** - to plot and analyze the performance of combination of genetic operators.

Deliverables

1. Analysing the convergence rate by using a varied combination of genetic operators.
2. Graphical Representation of comparisons.
3. Details report in addition to current literature reviewed by us.(given in report)

Final Presentation

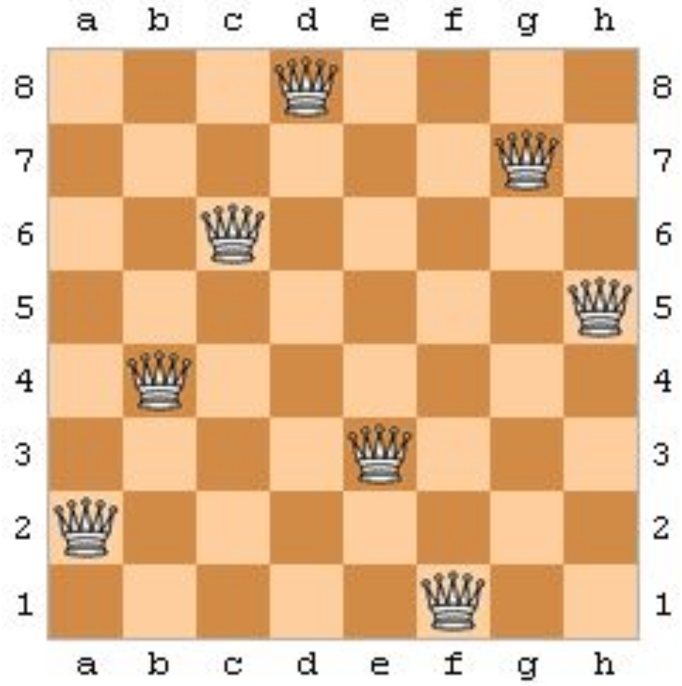
Constraint Handling

1. In order to prevent a single position from exceeding the boards limit. (Say 20X20 has 400 positions but after mutation the string returned is 11011110 then it exceeds 400), we have coded the mutations after an intervals such that it does not exceed the board limit.
2. Same problem occurred after crossmutations also, hence we coded the position in sets of different strings to avoid such an occurrence.
3. Eg ['01101101', '00001110', '10001101', '10000010', '00111001', '00101010']
4. Hence we did not have to attach any penalty to any solution.

Fitness Function

For each solution a fitness is defined as the number of blocks all the queens can cover on the board. The more the number of blocks a solution covers, the better it suits the problem statement.

In code:
def fitness(sol_list, n)
sol_list - a particular solution
n - board size



Roulette Wheel

We took the best ten solutions and assigned them the parents role on the basis of the roulette wheel which was biased towards allotting fitter parents.

```python
# 8. Runs the roulette wheel once on a set of ten solutions
# Solutions must be ordered in decreasing order of fitness
def run_roulette_wheel(best_ten):
    r = random.randint(0,360) #degeree of pointer after each spin
    chosen_two = []
    for i in range(15):
        if(0<=r<=140):
            chosen_two =[best_ten[0],best_ten[1]]
        elif(141<=r<=220):
            chosen_two =[best_ten[2],best_ten[3]]
        elif(221<=r<=286):
            chosen_two =[best_ten[4],best_ten[5]]
        elif(287<=r<=338):
            chosen_two =[best_ten[6],best_ten[7]]
        elif(339<=r<=360):
            chosen_two =[best_ten[8],best_ten[9]]
    return chosen_two
```

Elitist Selection

The best two chromosomes are preserved in the next generation.
This is done so that the best two solutions are never lost due to crossmutaions.

```
#12. Elitist Selection
# a,b are best solution of current generation
def elitist_selection(a,b):
    offsprings=[]

    #Preserving the parents(parents fed to this function are the best two of the generation)
    offsprings.append(a)
    offsprings.append(b)
    #single point crossover
    for i in range(5):
        x = deepcopy(b)
        y = deepcopy(a)
        r = random.randint(0,len(a)-1)
        x[:r] = a[:r]
        y[:r] = b[:r]
        #print(r)
        offsprings.append(x)
        offsprings.append(y)

    #double-point crossover
    for i in range(9):
        c = deepcopy(b)
        d = deepcopy(a)
        p = random.randint(1,len(b)-2)
        q  = random.randint(1,len(b)-2)
        c[p] = a[q]
        d[q] = b[p]
        offsprings.append(c)
        offsprings.append(d)

    return offsprings
```

Self-Made Selection

Very similar to elitist selection. However, here the two best are not preserved, they are used to derive the whole of the next generation with a mix of single point and double point cross-overs.

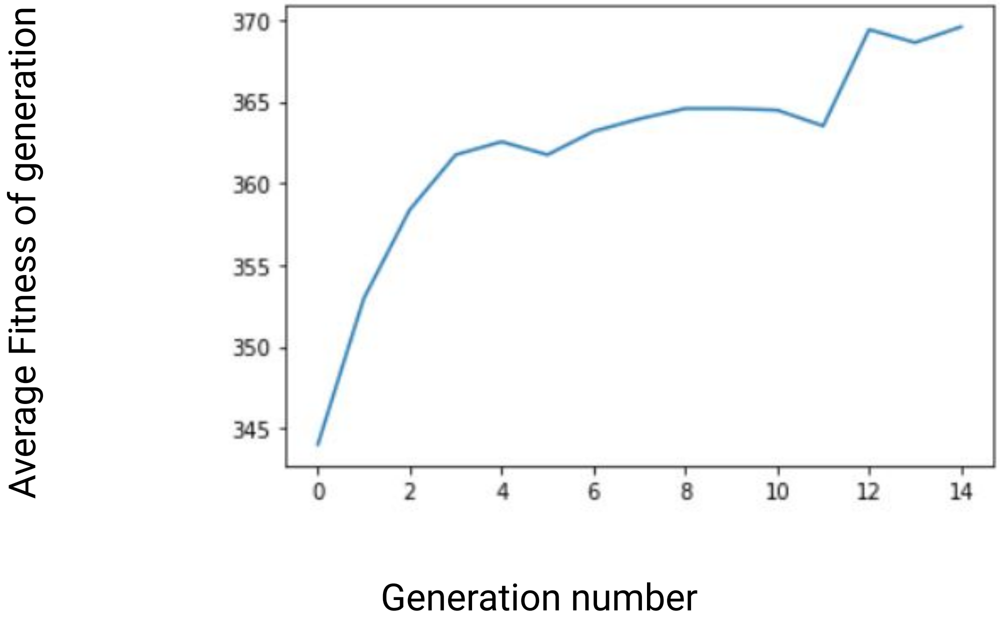
```python
#6. Crossmutation(a,b) - returns generation of size 30 with a,b as parent
# a - chosen parent chromosome
# b - chosen parent chromosome
def crossmutate_only_two_parents(a,b):
    offsprings=[]
    #single point crossover
    for i in range(5):
        x = deepcopy(b)
        y = deepcopy(a)
        r = random.randint(1,len(a)-1)
        x[:r] = a[:r]
        y[:r] = b[:r]
        #print(r)
        offsprings.append(x)
        offsprings.append(y)

    #double-point crossover
    for i in range(10):
        c = deepcopy(b)
        d = deepcopy(a)
        p = random.randint(0,len(b)-1)
        q  = random.randint(0,len(b)-1)
        c[p] = a[q]
        d[q] = b[p]
        offsprings.append(c)
        offsprings.append(d)

    return offsprings
```

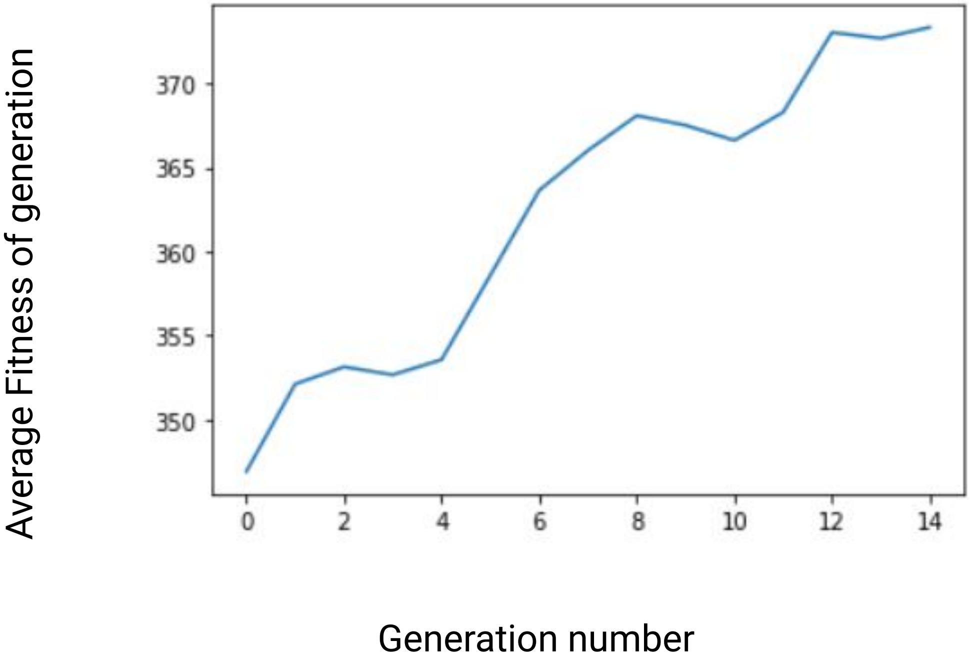
Results

Elitist Selection



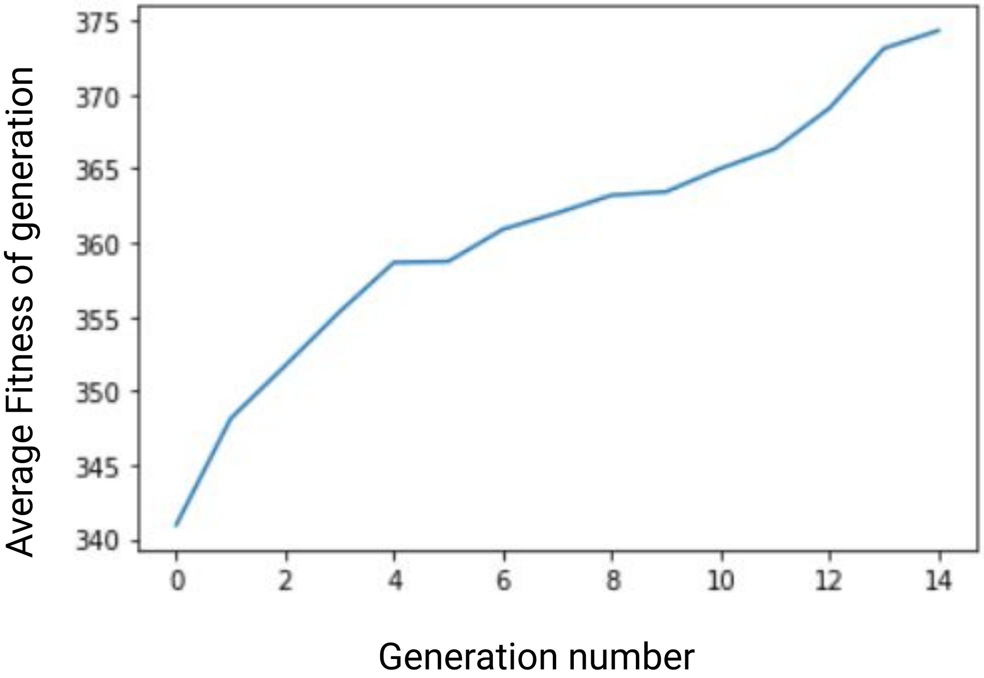
Results

Self-Made Selection



Results

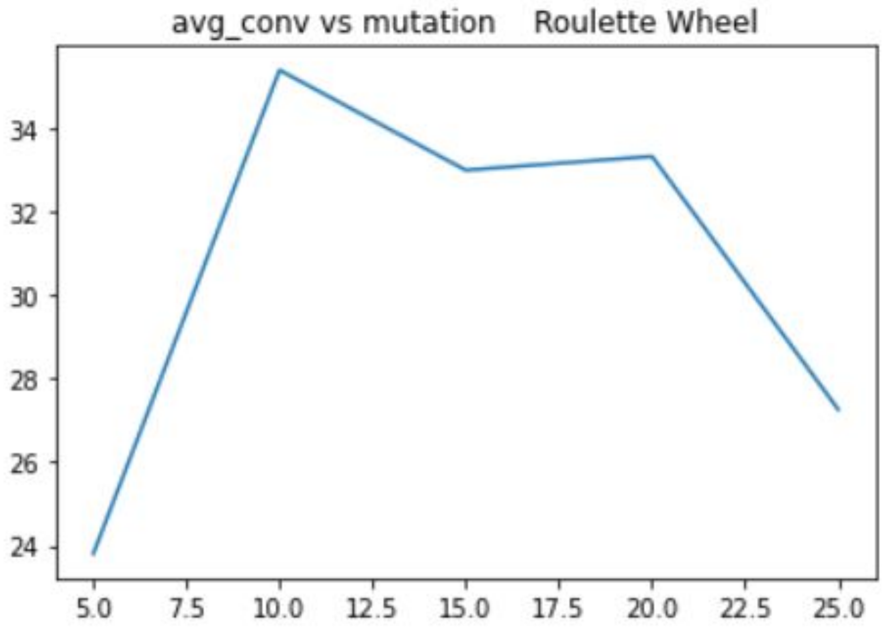
Roulette-Wheel Selection



Results

Roulette-Wheel Mutation

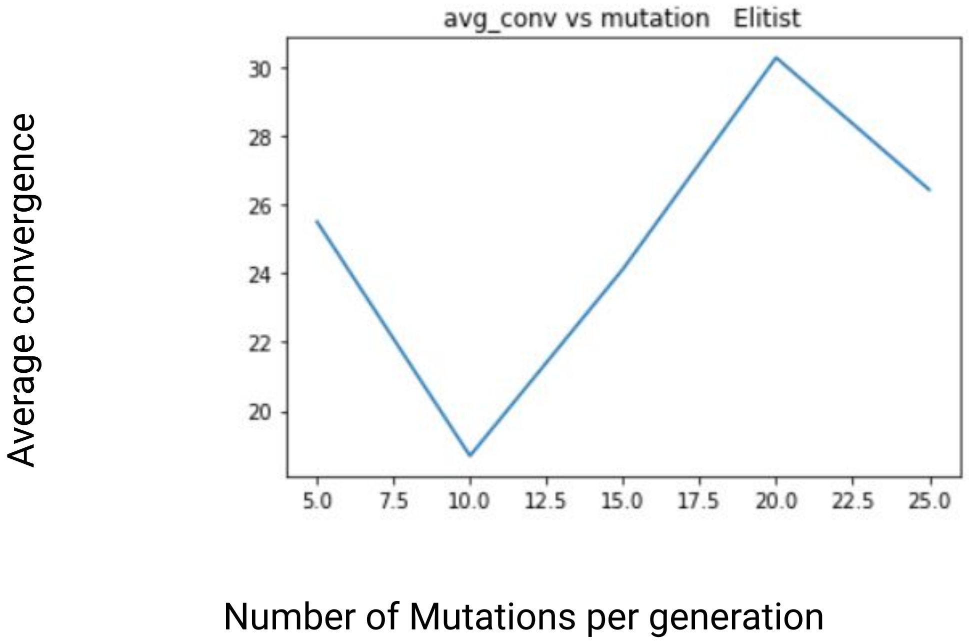
Average convergence



Number of mutations after each generation

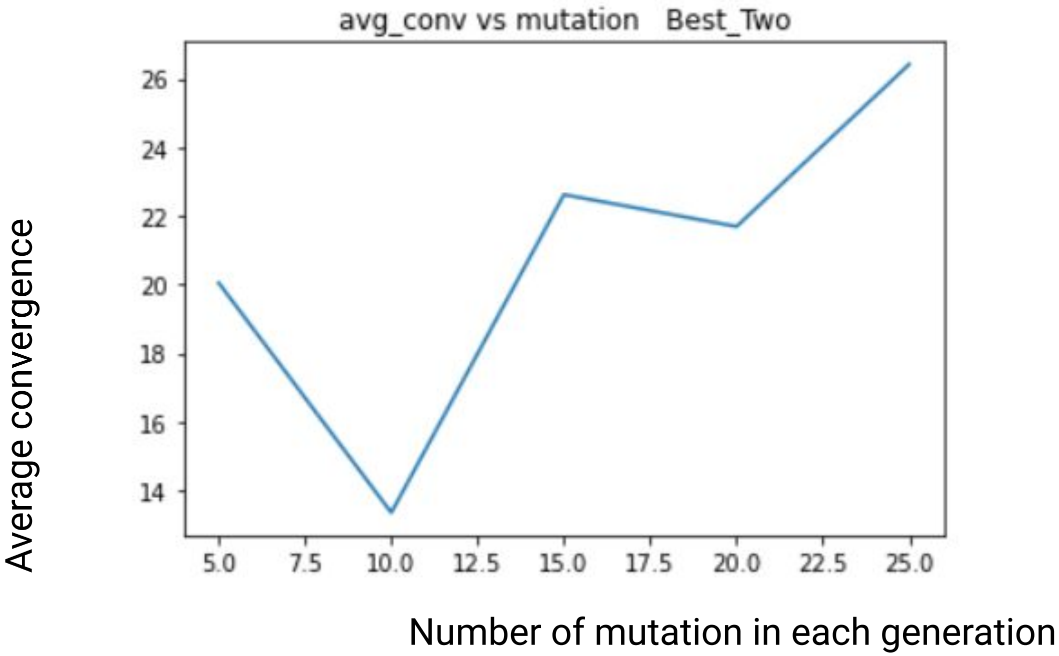
Results

Elitist Selection



Results

Self-Made Selection



References

Slide 2(image)
https://www.semanticscholar.org/paper/A-Genetic-Algorithm-Based-Approach-for-Solving-the-Alharbi-Venkat/216290df1c920d8a68bcbda7600ae93add6ac4a5

Slide 4(image)
https://www.researchgate.net/publication/309770246_A_Study_on_Genetic_Algorithm_and_its_Applications)

Slide 6-10(image)
https://www.tutorialspoint.com/genetic_algorithms

Thank you