كلية علوم الحاسب والمعلومات
قسم تقنية المعلومات

King Saud University
College of Computer and Information Sciences
Department of Information Technology

IT426: ARTIFICIAL INTELLIGENCE SYSTEMS

ASSIGNMENT 3

Genetic algorithm Report

Solution representation

Team work:

Raghad Althunayan:437202263
Haifa Alkhudair :437200221

For every piece of outfit we created dataframe using Python pandas library (top, bottom, shoes, necklace, handbag). Example (top):

	CATEGORY	PIECE	COLOR	DRESS_CODE	PRICE	ID
0	TOP	t-shirt	dark	casual	0	1
1	TOP	t-shirt	dark	sportswear	0	2
2	TOP	t-shirt	bright	casual	0	3
3	TOP	t-shirt	bright	sportswear	0	4
4	TOP	blouse	dark	business	200	5
5	TOP	blouse	dark	evening	200	6
6	TOP	bodysuit	dark	casual	150	7
7	TOP	bodysuit	dark	evening	150	8
8	TOP	sleeveless	bright	casual	110	9
9	TOP	sweater	dark	casual	200	10

Every row has its index. We create population by selecting random index for each pieces of outfit. One row in population dataframe represents one solution. Example (population):

	TOP	BOTTOM	SHOES	NECK	HANDBAG	FITNESS
0	[17]	[11]	[1]	[7]	[1]	0.24
1	[19]	[5]	[1]	[6]	[4]	0.40
2	[16]	[8]	[10]	[4]	[5]	0.24
3	[15]	[10]	[1]	[3]	[5]	0.12
4	[20]	[4]	[1]	[7]	[4]	0.28
5	[8]	[1]	[10]	[4]	[5]	0.32
6	[5]	[8]	[7]	[7]	[3]	0.12
7	[7]	[11]	[4]	[1]	[1]	0.36
8	[10]	[11]	[3]	[3]	[5]	0.40
9	[8]	[6]	[6]	[3]	[5]	0.36

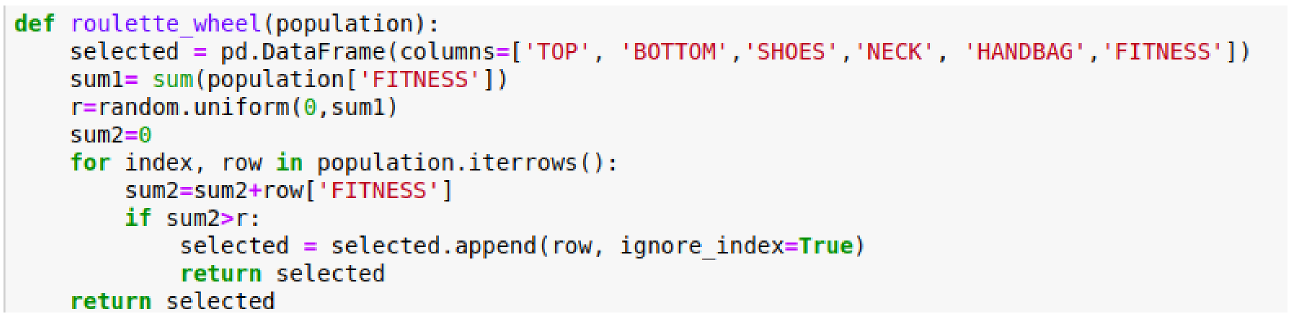
After generating columns 'TOP', 'BOTTOM', 'SHOES', 'NECK' and 'HANDBAG' program can calculate 'FITNESS' value for every row. Below we can see example of formatted solution:

```
TOP: ['cardigan', 'dark', 'casual', 300]
BOTTOM: ['legging ', 'bright', 'casual', 100]
SHOES: ['flat ', 'dark', 'casual', 0]
NECK: ['choker ', 'bright', 'casual', 0]
HANDBAG: ['cross bag ', 'dark', 'business', 0]
FITNESS: 0.8400000000000001
```

Fitness Function

We calculate color fitness value by counting how many pieces match and dividing it by 5. We did same thing works for dresscode. For budget We checked if there is enough money for whole outfit. If there is then fitness is 1 else fitness is 0. We transform three fitness values into one final fitness value by multiplying color fitness by 0.2, dresscode fitness by 0.4 and budget fitness by 0.4.

Roulette wheel selection



```python
def roulette_wheel(population):
    selected = pd.DataFrame(columns=['TOP', 'BOTTOM','SHOES','NECK', 'HANDBAG','FITNESS'])
    sum1= sum(population['FITNESS'])
    r=random.uniform(0,sum1)
    sum2=0
    for index, row in population.iterrows():
        sum2=sum2+row['FITNESS']
        if sum2>r:
            selected = selected.append(row, ignore_index=True)
            return selected
    return selected
```

In the code above we can see the implementation of roulette wheel selection. First part of function is calculating sum of all fitness values in the population. Then we choose random uniform value from 0 to sum1 where sum1 is previous calculated sum. Then we search for first point where cumulative sum is higher than the random uniform value from 0 to sum1. we return row (solution) with that index.

Crossover

```python
LABELS = ['TOP', 'BOTTOM', 'SHOES', 'NECK', 'HANDBAG']
N = len(LABELS)

def crossover(parent1, parent2, color, code, budget, crossover_rate = 1.0):
    u = random.random()

    # If the dataframes are not copied then we can have some unexpected results
    # We must make copies!
    child1 = parent1.copy()
    child2 = parent2.copy()

    # u is random value from range: [0, 1]
    # if u is smaller than crossover rate probability
    # then we apply crossover operator to parents and
    # otherwise, crossover operator is not applied and
    # same solutions are returned (parent1 == child1, parent2 == child2)
    if u <= crossover_rate:
        point = random.randrange(0, N-1)
        for i in range(0, N):
            label = LABELS[i]

            if i < point:
                child1[label] = parent2[label]
                child2[label] = parent1[label]
            else:
                child1[label] = parent1[label]
                child2[label] = parent2[label]

    # We must recalculate fitness!
    child1['FITNESS'] = fit(child1,color,code,budget,top,bottom,shoes,neck,handbag)
    child2['FITNESS'] = fit(child2,color,code,budget,top,bottom,shoes,neck,handbag)

    return child1, child2
```

We implemented one-point crossover by generating random point from 0 to 5 (5 pieces) and then swapping all pieces from 0 to point(random value). we apply crossover operator with default probability 1.0 or crossover_rate probability if argument is set in function call. Below we can see example of one-point crossover:

```
parents:
    TOP BOTTOM SHOES NECK HANDBAG  FITNESS
0  [19]     [4]    [9]  [3]      [5]     0.84
   TOP BOTTOM SHOES NECK HANDBAG  FITNESS
0  [4]     [4]    [3]  [7]      [5]     0.56
children:
   TOP BOTTOM SHOES NECK HANDBAG  FITNESS
0  [4]     [4]    [9]  [3]      [5]     0.72
    TOP BOTTOM SHOES NECK HANDBAG  FITNESS
0  [19]     [4]    [3]  [7]      [5]     0.28
```

Mutation

```
LABELS = ['TOP', 'BOTTOM', 'SHOES', 'NECK', 'HANDBAG']
PIECES = [top, bottom, shoes, neck, handbag]
N = len(PIECES)

def mutate(unit, color, code, budget, mutation_rate = 0.01):
    result = unit.copy()

    # if u is higher than mutation rate then we do not apply mutation
    u = random.random()
    if u >= mutation_rate:
        return result

    # mutation is done by replacing random piece of outfit
    # with another piece of outfit
    point = random.randrange(0, N)
    piece = PIECES[point]
    label = LABELS[point]
    new_sample = piece.sample()

    result[label] = [new_sample['ID'].values]

    # we must recalculate fitness
    result['FITNESS'] = fit(child,color,code,budget,top,bottom,shoes,neck,handbag)
    return result
```

We implemented mutation by replacing random piece of outfit with another new piece of outfit. Below we can see mutation example(first row is not mutated and second row is mutated):

```
    TOP BOTTOM SHOES NECK HANDBAG  FITNESS
0  [19]    [1]   [4]  [1]     [1]     0.36
    TOP BOTTOM SHOES NECK HANDBAG  FITNESS
0  [19]    [5]   [4]  [1]     [1]     0.36
```

Replacement

We replace whole population every generation. In considered implementation of replacement with elitism where we make sure that some of the best solutions (example: 10%) is passed directly to new generation.

Analysis of my results

Implementation of genetic algorithm is not done when we finish the code implementation. Finding good parameters for GA is next part of the task. Below we can see list of graphs plots with combination of parameters for:

- mutation rate: [0.001, 0.03, 0.5]
- crossover rate: [0.05, 0.9, 1.0]
- population size: [10, 20, 30]

We want to find optimal parameters.

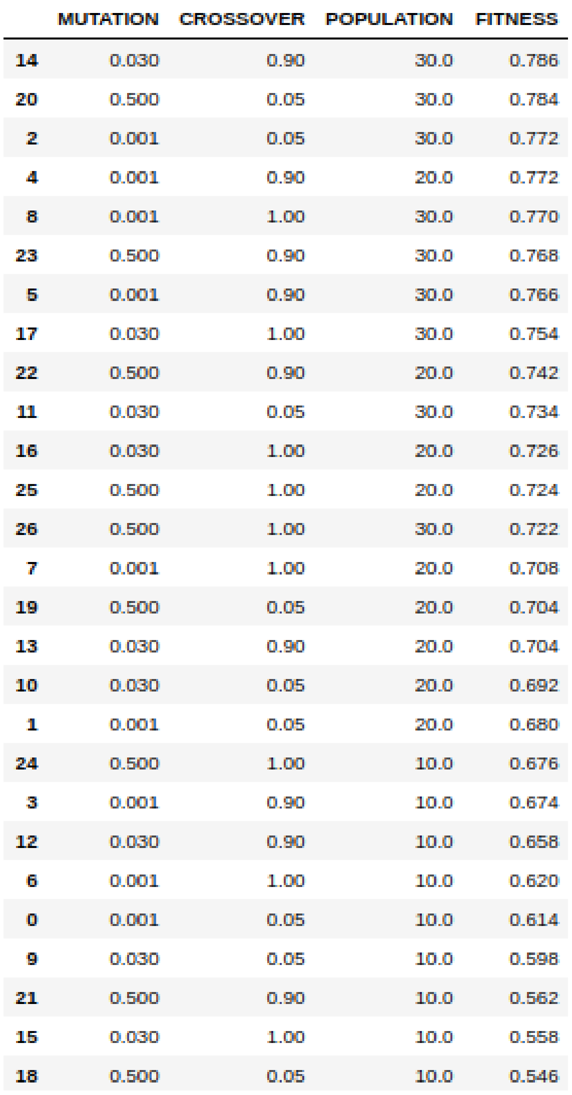
Another parameter that can be discussed that is not in the list above is maximal iteration parameter for termination condition. Increasing maximal iteration parameter loses its value from some point. We can see that on graphs below. Difference between first and 10th generation best fitnesses, and 10th before last and last generation best fitnesses is a lot higher. We can see improvement function best solution by parameter for maximal iteration as a logarithmic function.

Easiest parameter to analyze from graphs is population size. Intuitively, we can expect better result from bigger population size because we explore more space in search with bigger population. In this case this assumption is positive. We can see that GA with population 30 performs a lot better than GA with population 10.

From graphs below we can see that algorithm with high mutation rate are unstable and have big spikes. With higher mutation rate we have better search space exploration but with cost of losing good solutions by mutating it (we are not using elitism here). We can see that GA with 0.5 is unstable (mostly in first few generations) where GA with 0.001 is really stable.

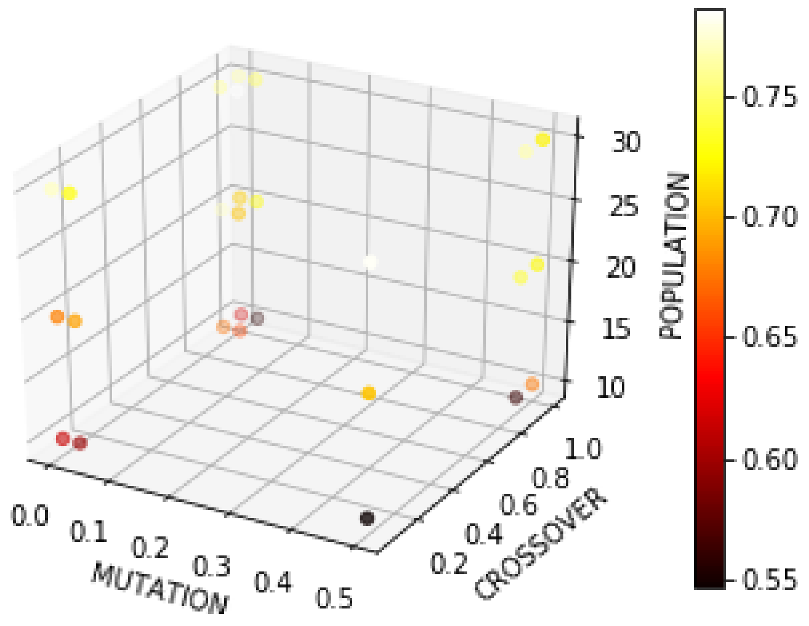
For crossover parameter we can’t see that much difference on graphs between 0.9 and 1.0 crossover rates. We can see some difference between those 2 and 0.05 crossover rate where 0.9 and 1.0 perform a bit better.

Average fitness by parameter combination:

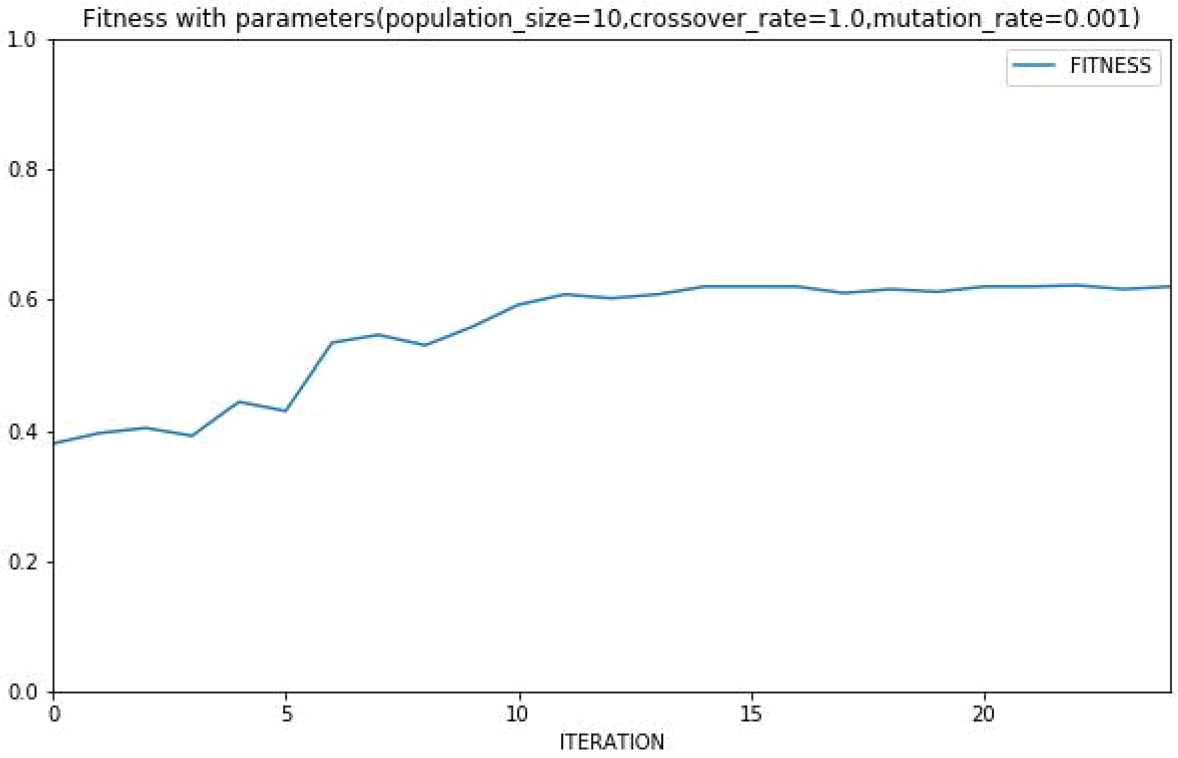


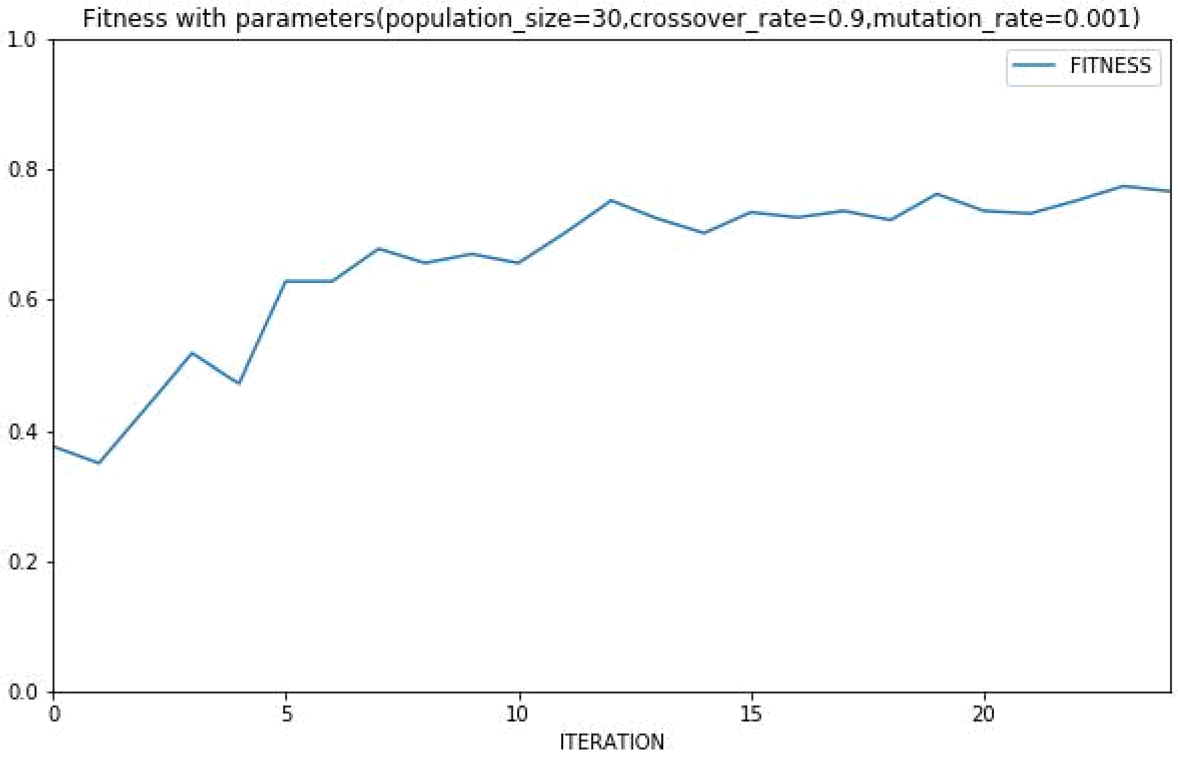
	MUTATION	CROSSOVER	POPULATION	FITNESS
14	0.030	0.90	30.0	0.786
20	0.500	0.05	30.0	0.784
2	0.001	0.05	30.0	0.772
4	0.001	0.90	20.0	0.772
8	0.001	1.00	30.0	0.770
23	0.500	0.90	30.0	0.768
5	0.001	0.90	30.0	0.766
17	0.030	1.00	30.0	0.754
22	0.500	0.90	20.0	0.742
11	0.030	0.05	30.0	0.734
16	0.030	1.00	20.0	0.726
25	0.500	1.00	20.0	0.724
26	0.500	1.00	30.0	0.722
7	0.001	1.00	20.0	0.708
19	0.500	0.05	20.0	0.704
13	0.030	0.90	20.0	0.704
10	0.030	0.05	20.0	0.692
1	0.001	0.05	20.0	0.680
24	0.500	1.00	10.0	0.676
3	0.001	0.90	10.0	0.674
12	0.030	0.90	10.0	0.658
6	0.001	1.00	10.0	0.620
0	0.001	0.05	10.0	0.614
9	0.030	0.05	10.0	0.598
21	0.500	0.90	10.0	0.562
15	0.030	1.00	10.0	0.558
18	0.500	0.05	10.0	0.546

This 3d plot of data above:

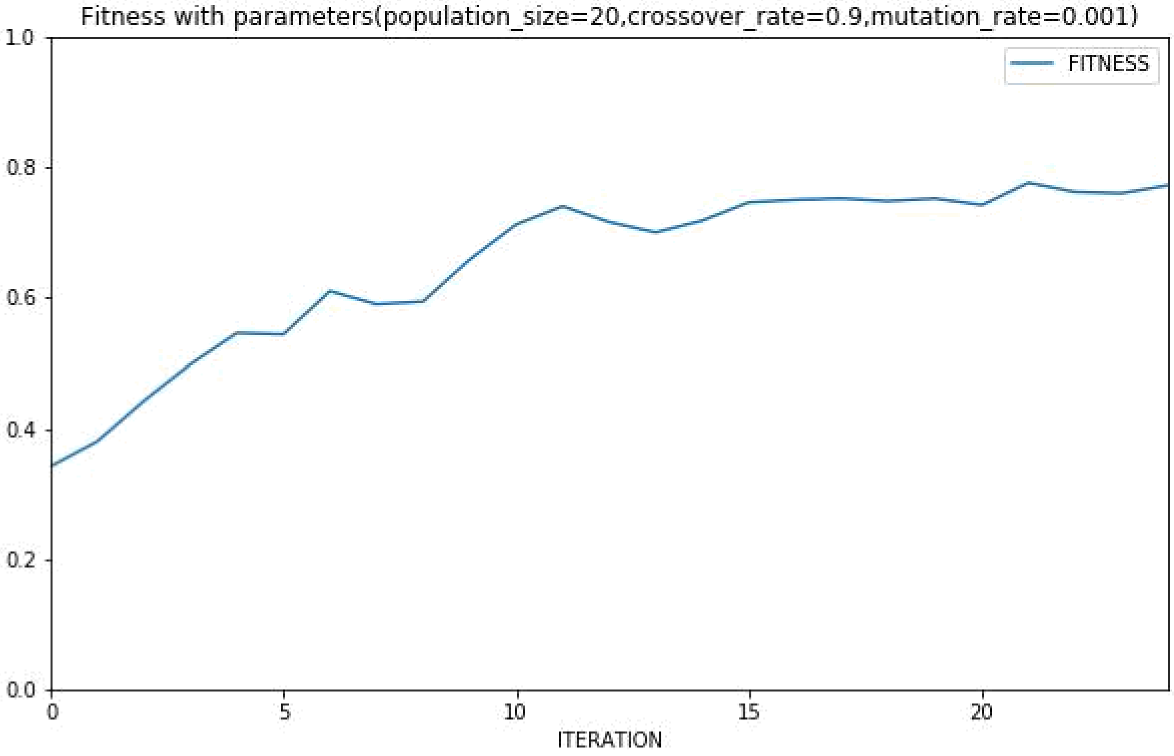


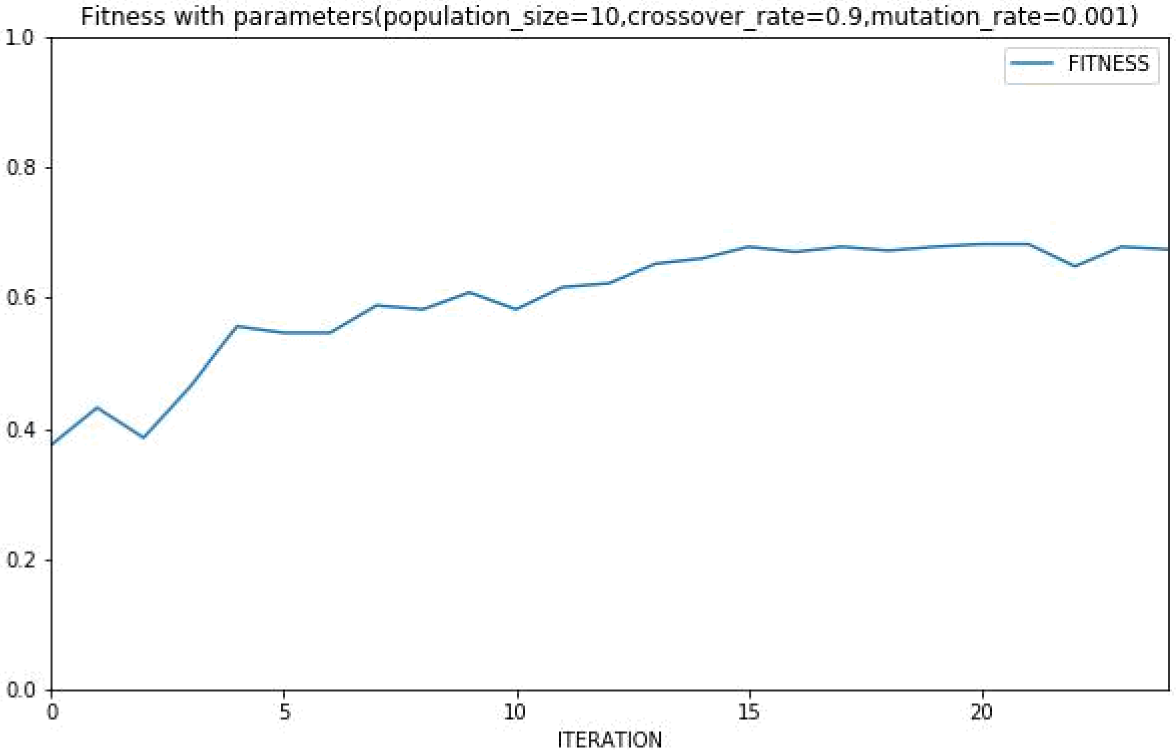
Plots are generated using pandas library:

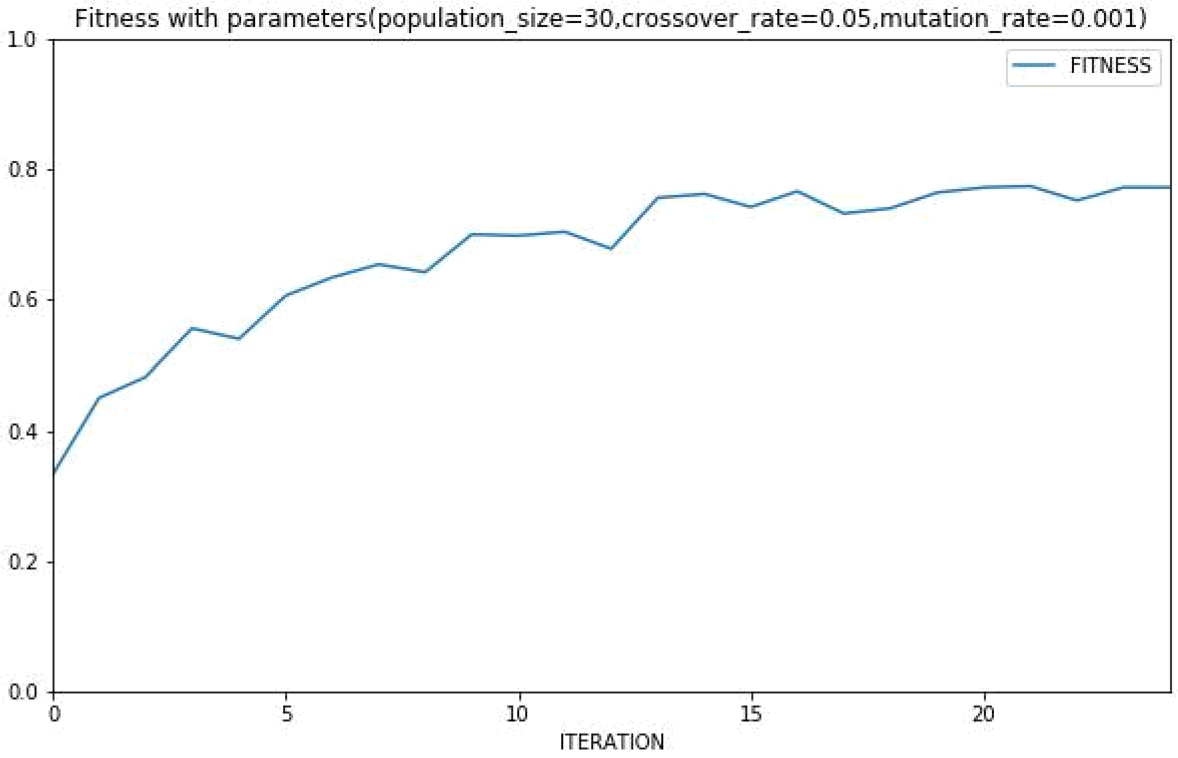

Fitness with parameters(population_size=10,crossover_rate=1.0,mutation_rate=0.001)
FITNESS
ITERATION
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20

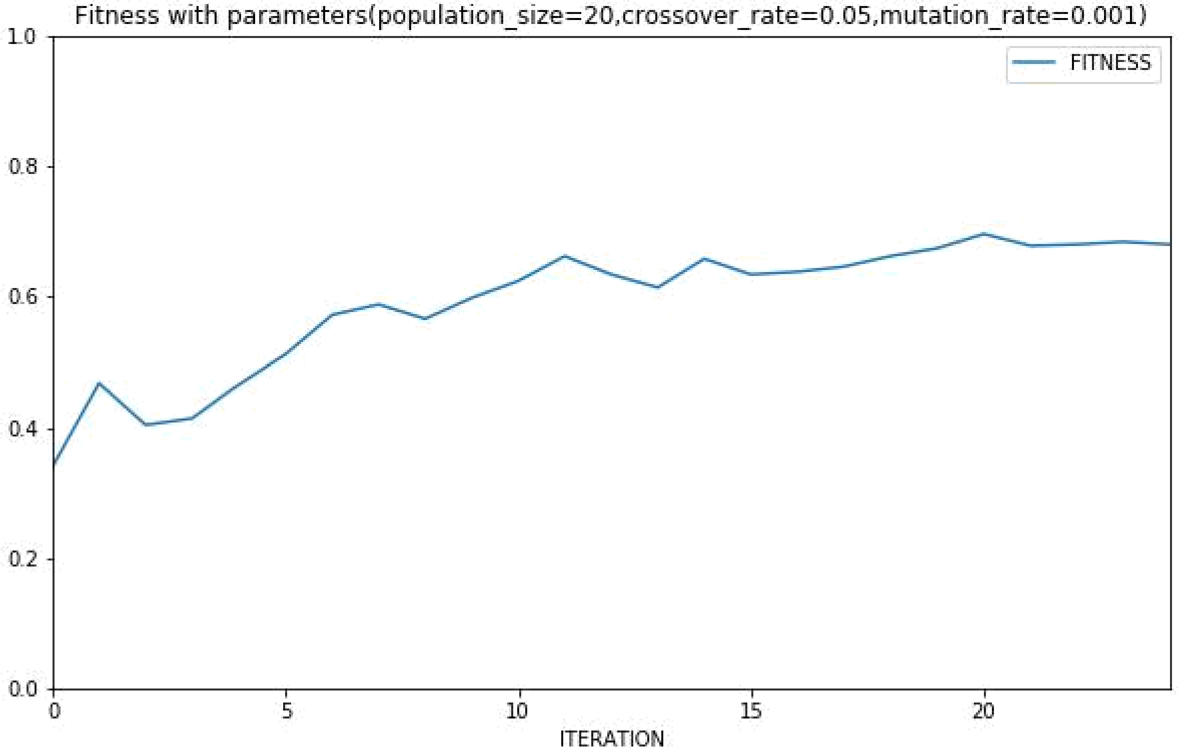


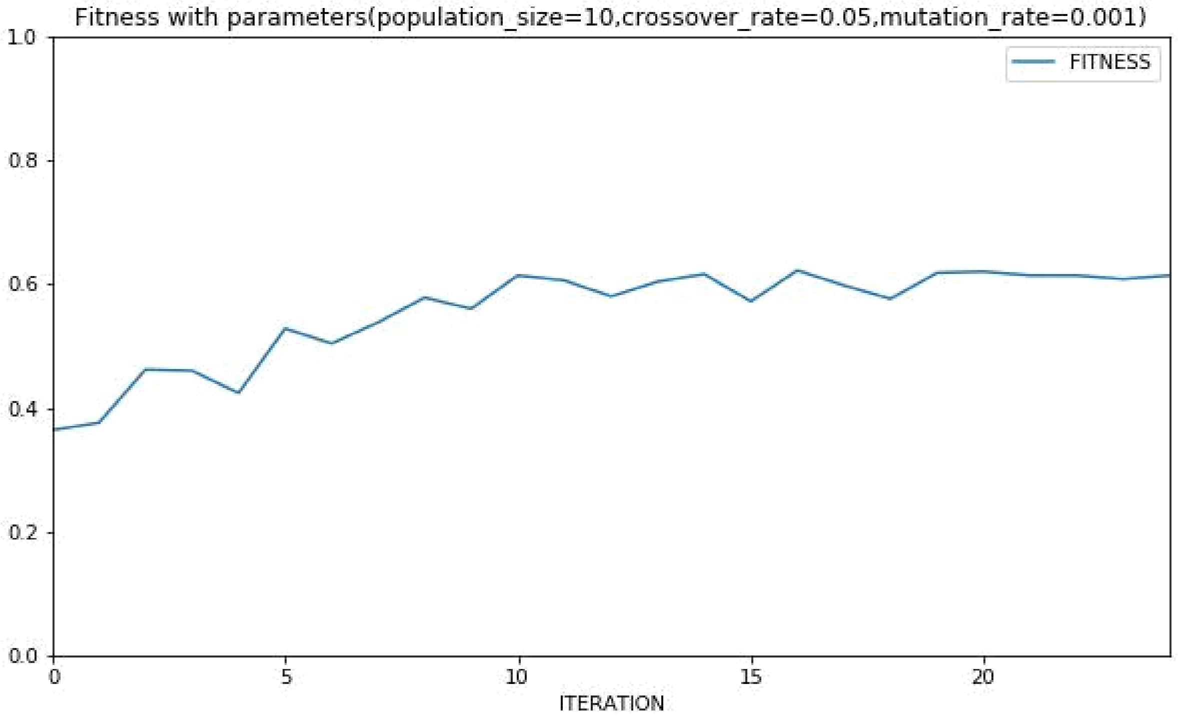
Fitness with parameters(population_size=30,crossover_rate=0.9,mutation_rate=0.001)
FITNESS
ITERATION
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20

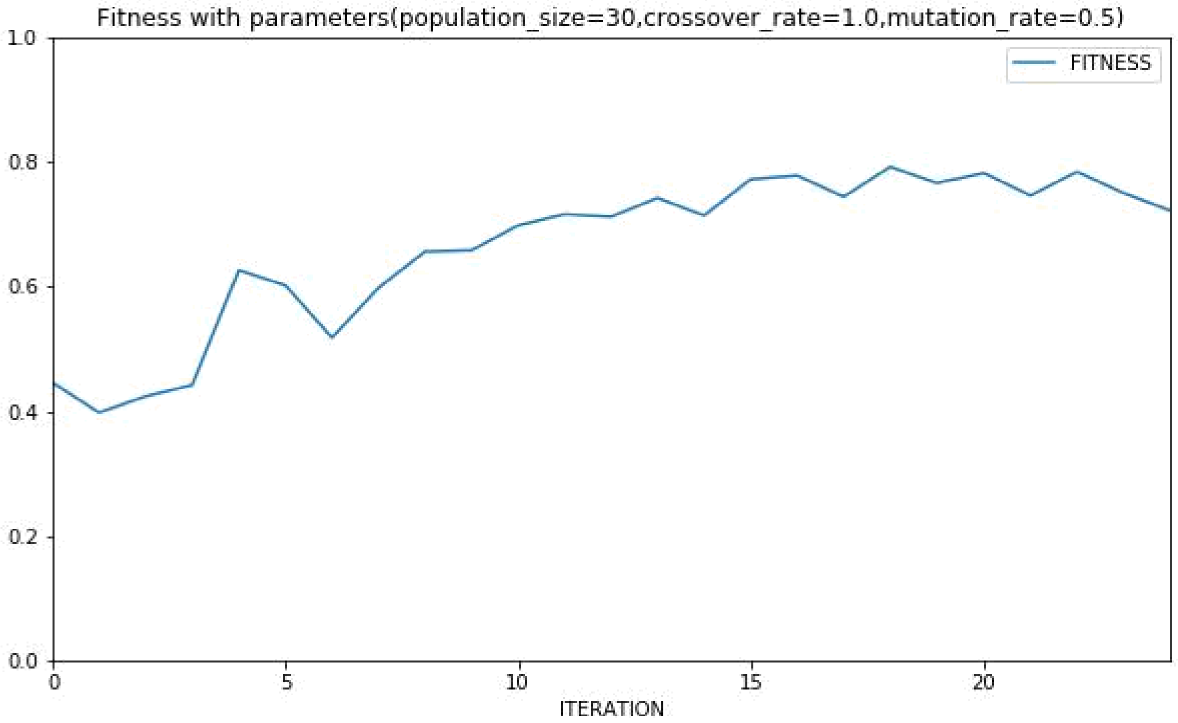

Fitness with parameters(population_size=20,crossover_rate=0.9,mutation_rate=0.001)
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
ITERATION
FITNESS

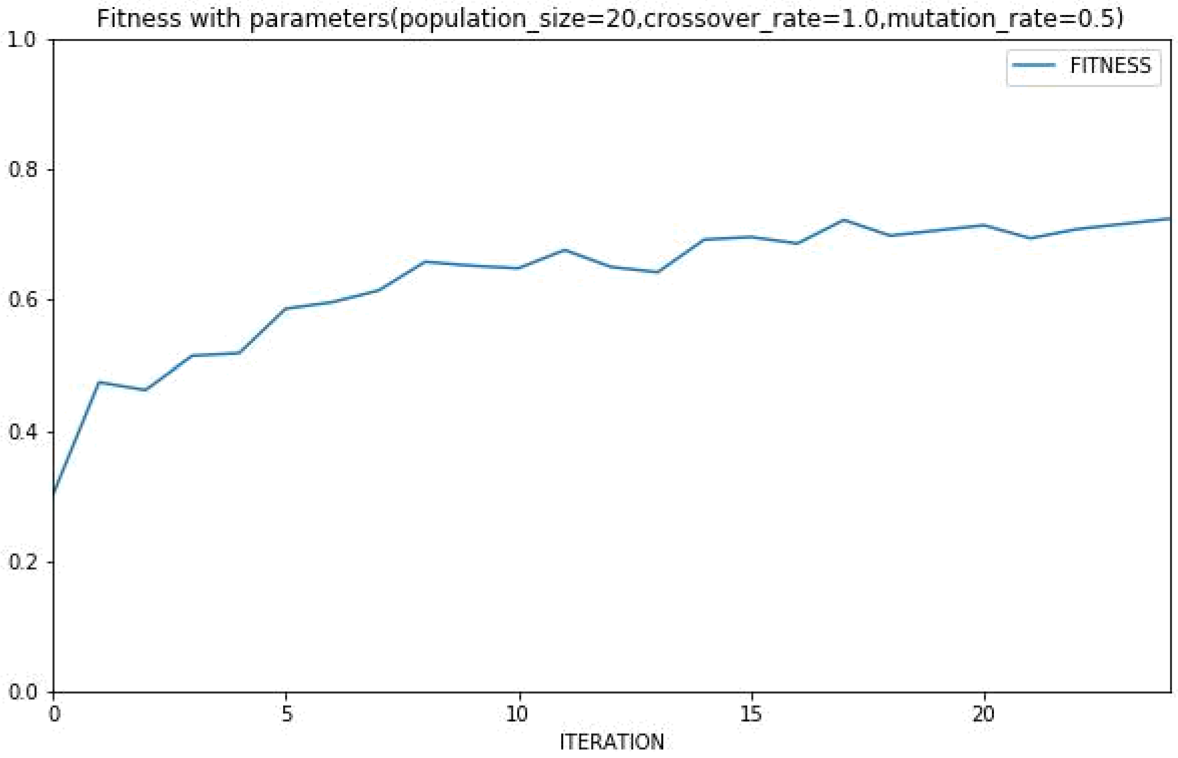

Fitness with parameters(population_size=10,crossover_rate=0.9,mutation_rate=0.001)
FITNESS
ITERATION
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20

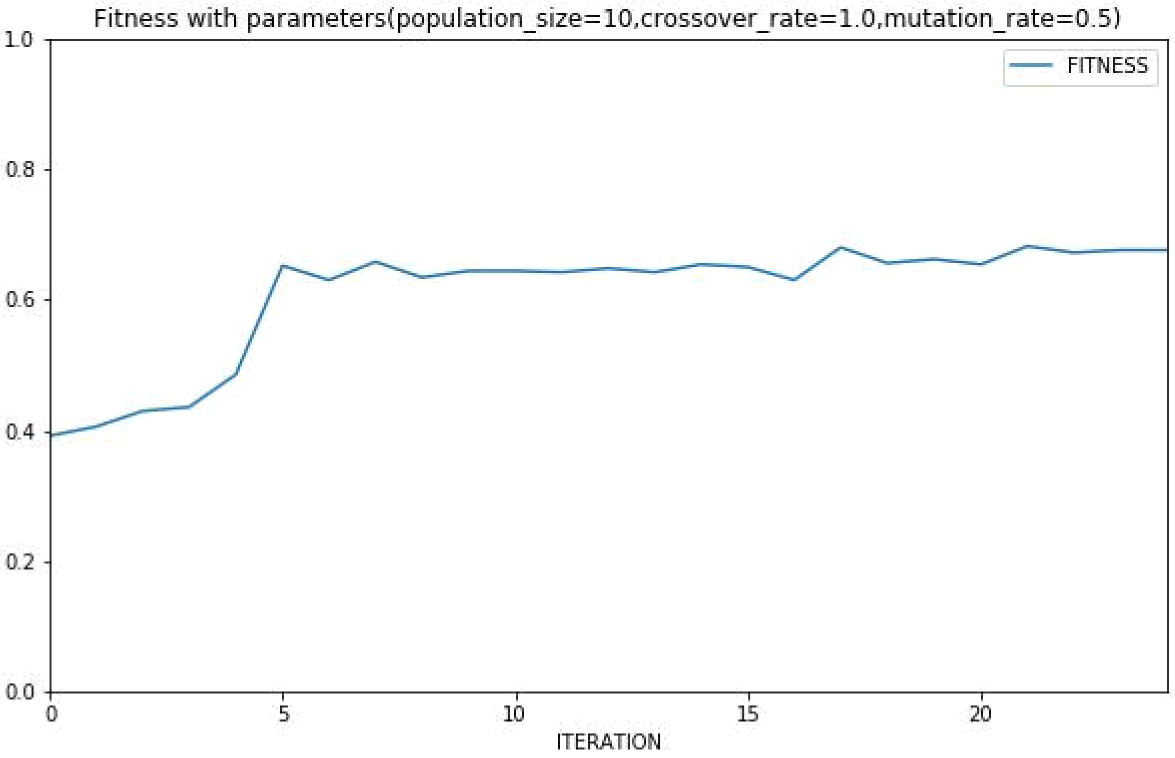

Fitness with parameters(population_size=30,crossover_rate=0.05,mutation_rate=0.001)
FITNESS
ITERATION
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20

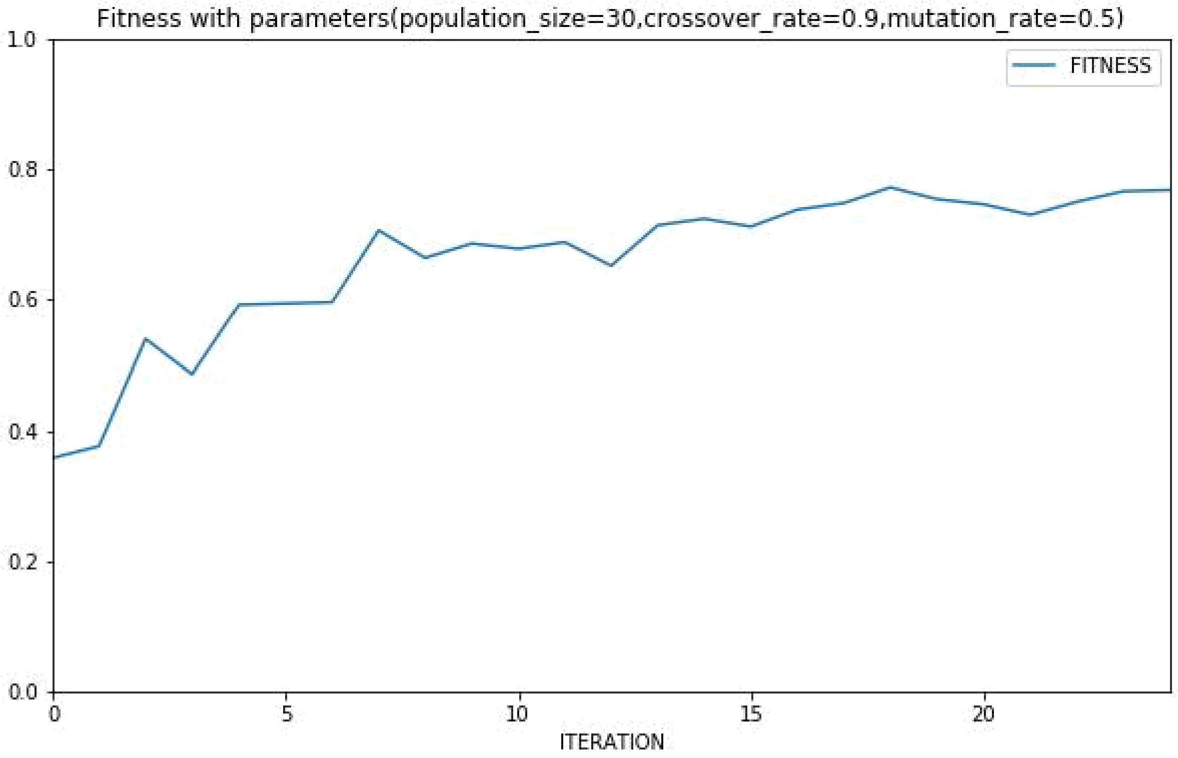

Fitness with parameters(population_size=20,crossover_rate=0.05,mutation_rate=0.001)
FITNESS
ITERATION
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20

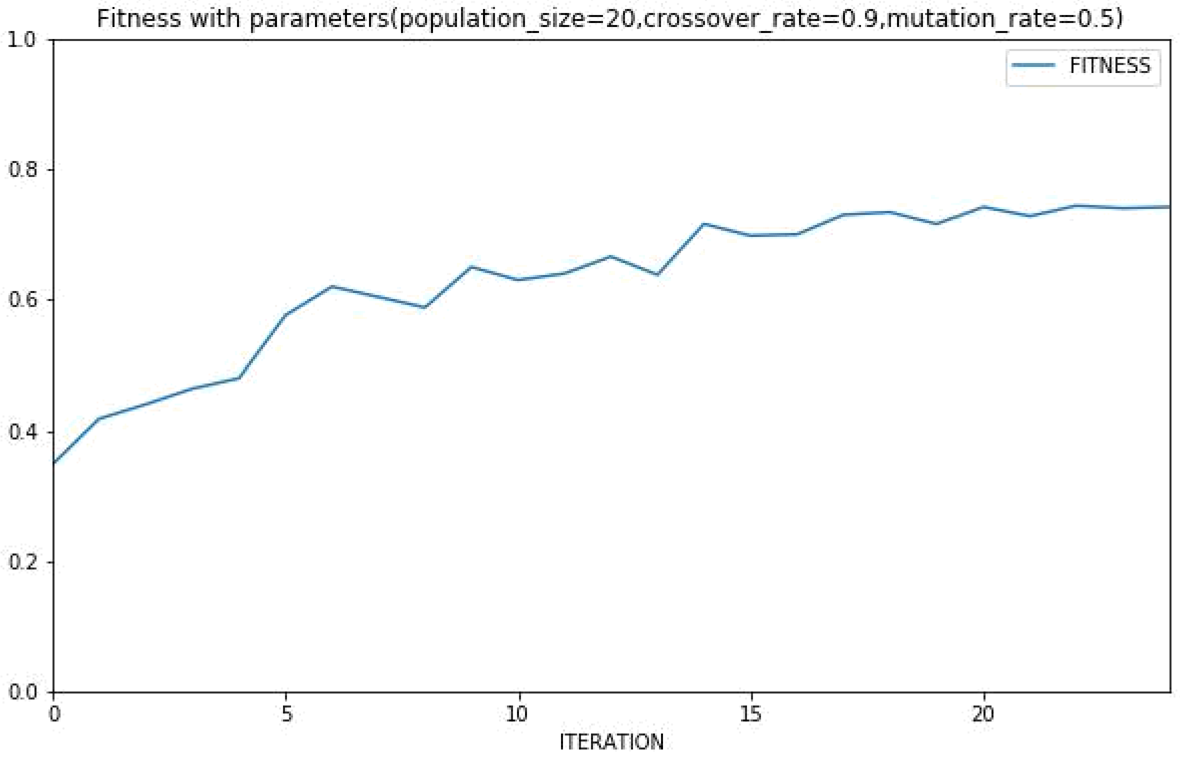

Fitness with parameters(population_size=10,crossover_rate=0.05,mutation_rate=0.001)
FITNESS
ITERATION
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20


Fitness with parameters(population_size=30,crossover_rate=1.0,mutation_rate=0.5)
FITNESS
ITERATION
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20

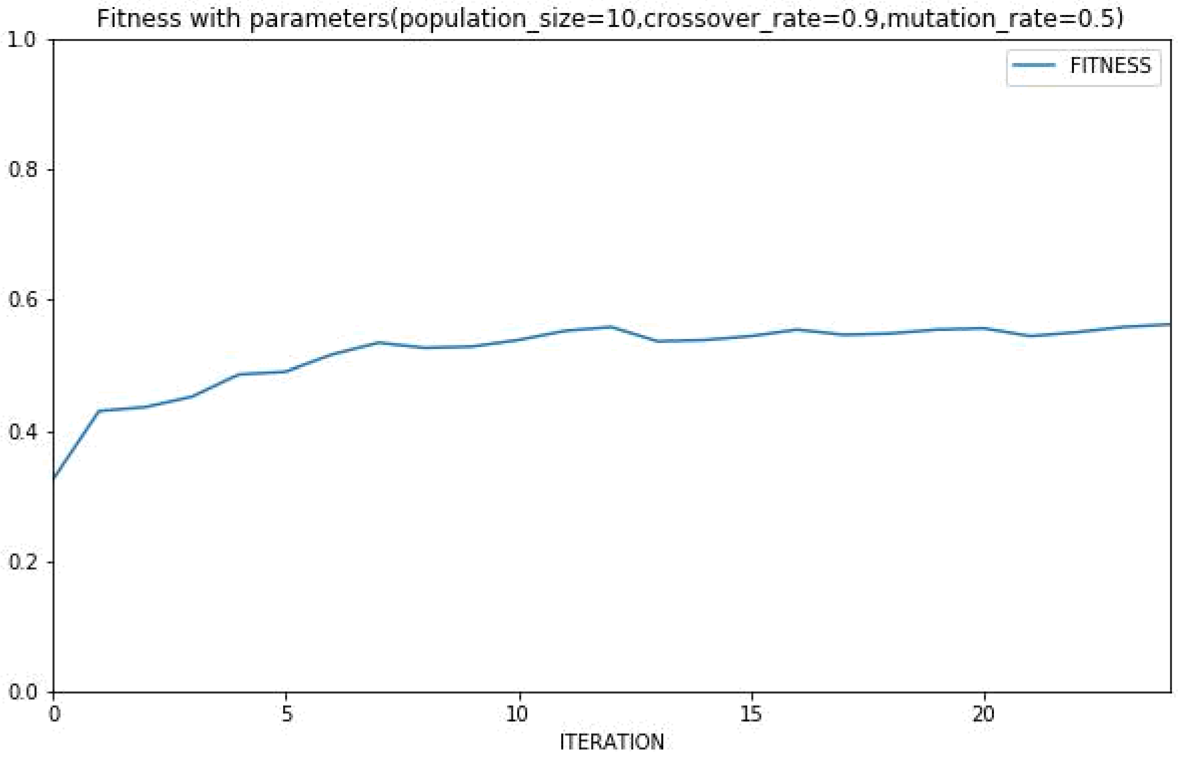

Fitness with parameters(population_size=20,crossover_rate=1.0,mutation_rate=0.5)
FITNESS
ITERATION
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20

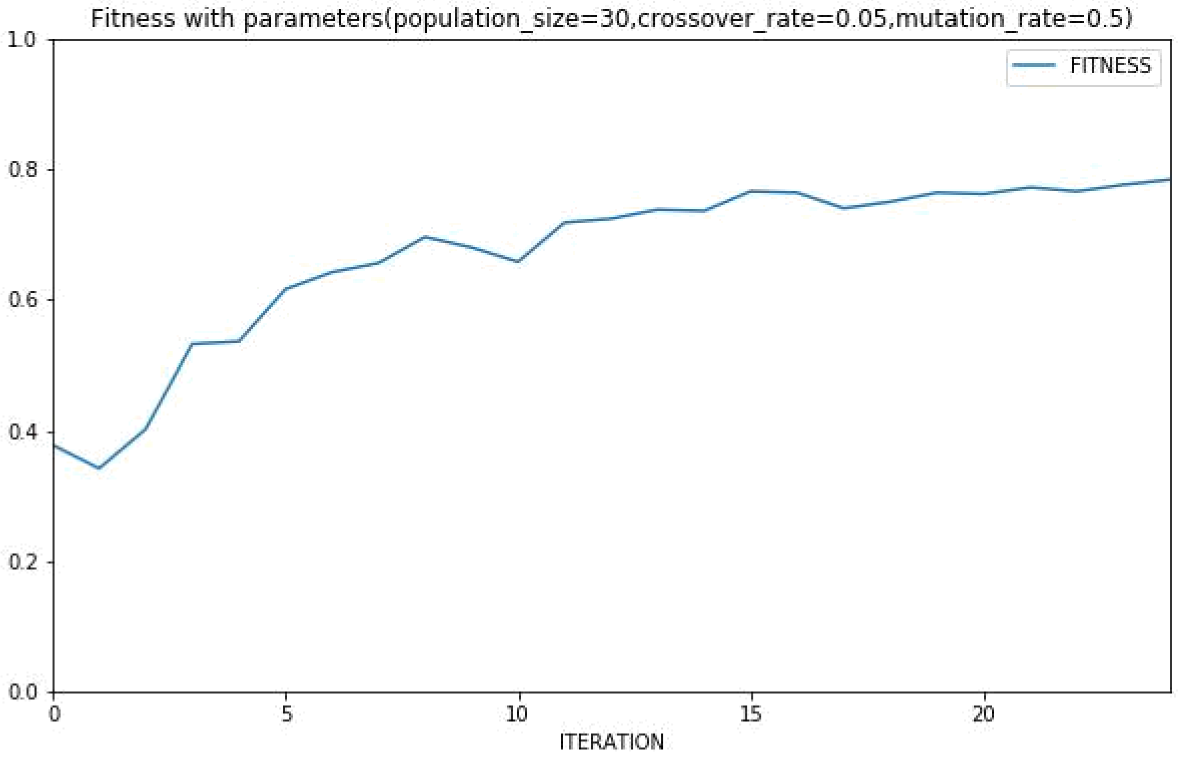

Fitness with parameters(population_size=10,crossover_rate=1.0,mutation_rate=0.5)
FITNESS
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
ITERATION

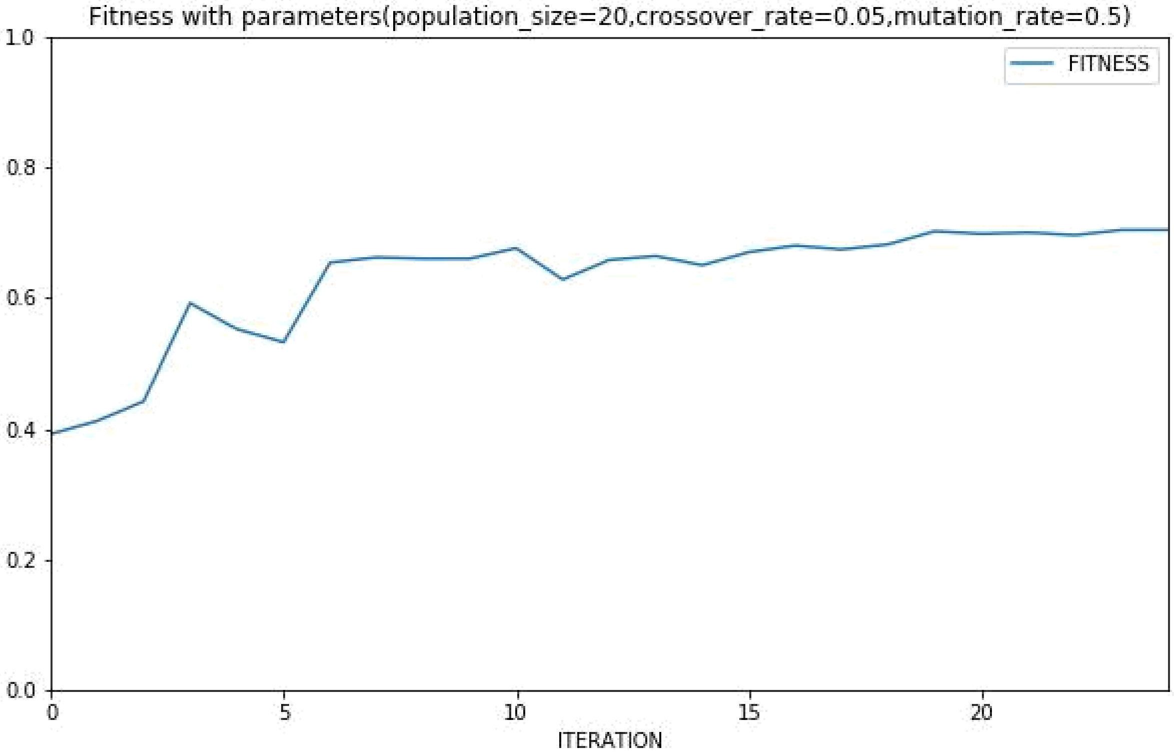

Fitness with parameters(population_size=30,crossover_rate=0.9,mutation_rate=0.5)
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
ITERATION
FITNESS

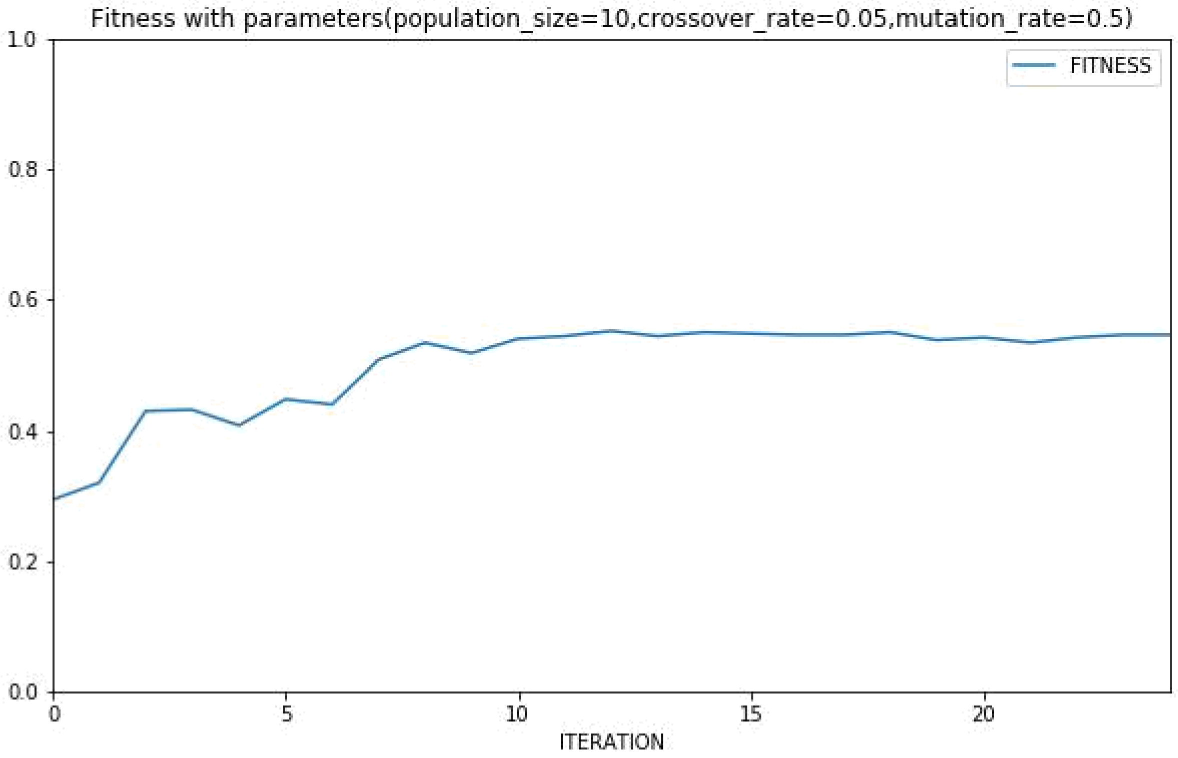


Fitness with parameters(population_size=20,crossover_rate=0.9,mutation_rate=0.5)
FITNESS
ITERATION
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20

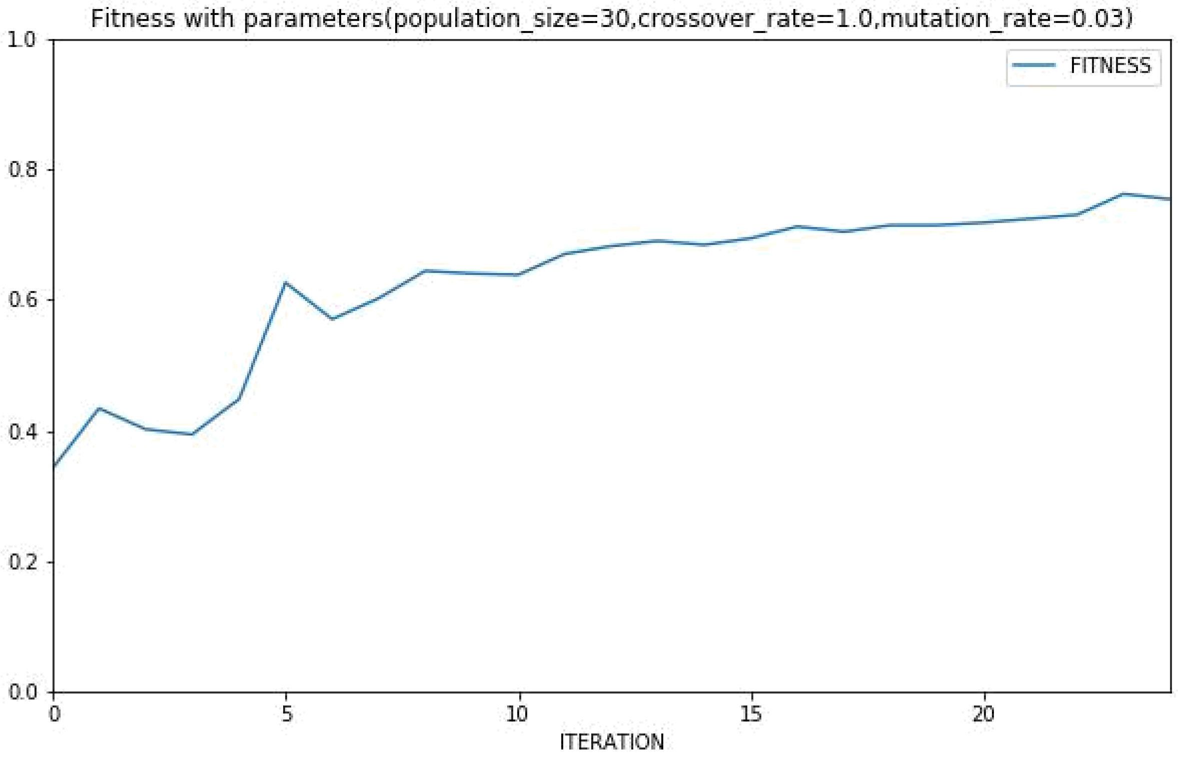

Fitness with parameters(population_size=10,crossover_rate=0.9,mutation_rate=0.5)
FITNESS
ITERATION
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20

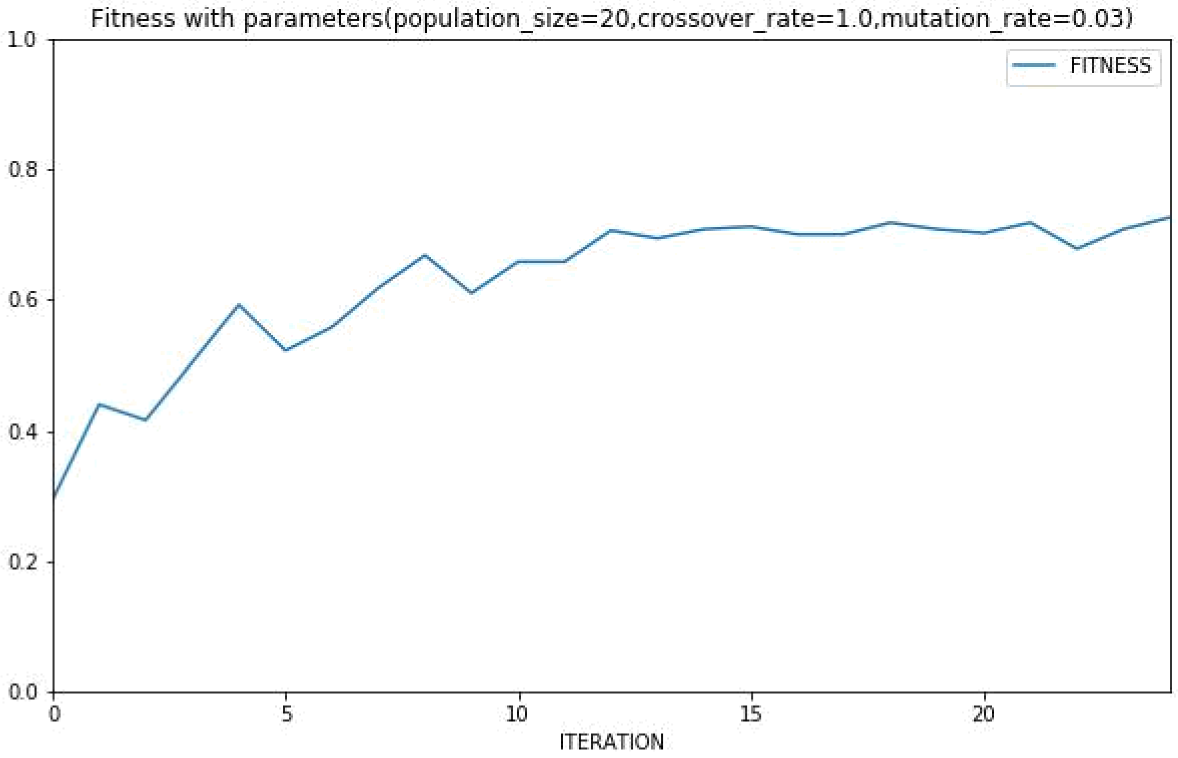

Fitness with parameters(population_size=30,crossover_rate=0.05,mutation_rate=0.5)
FITNESS
ITERATION
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20


Fitness with parameters(population_size=20,crossover_rate=0.05,mutation_rate=0.5)
FITNESS
ITERATION
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20

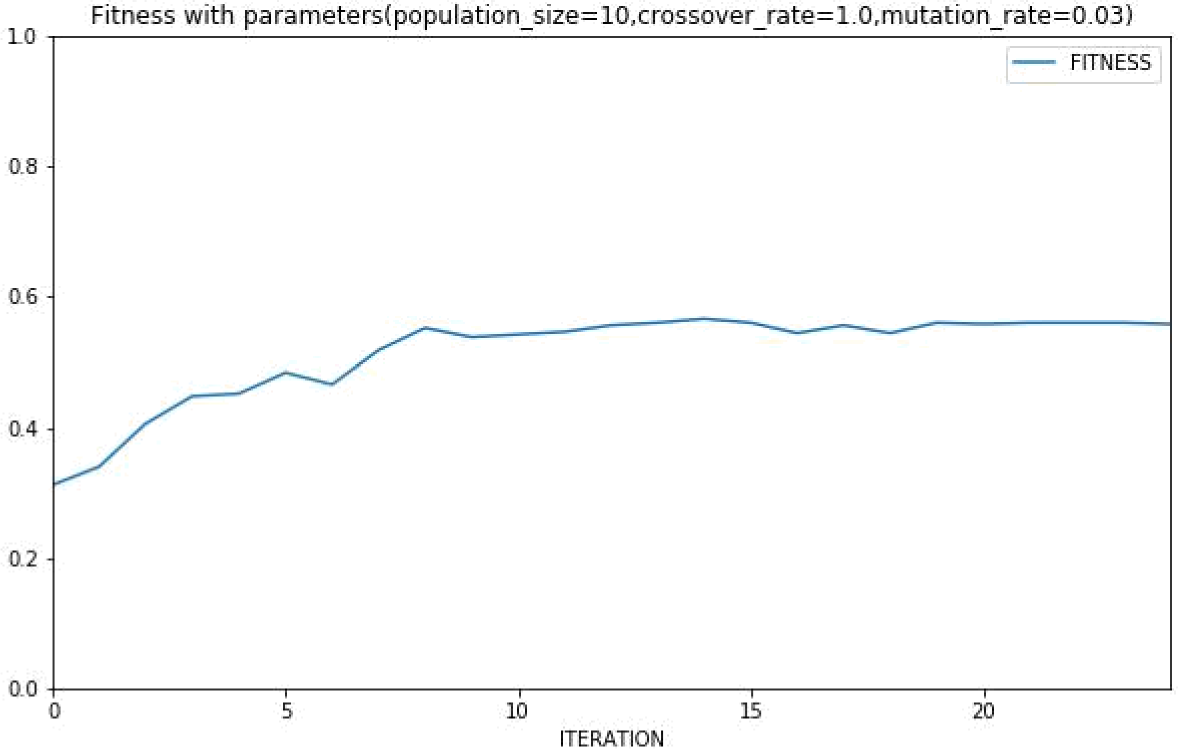


Fitness with parameters(population_size=10,crossover_rate=0.05,mutation_rate=0.5)
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
ITERATION
FITNESS

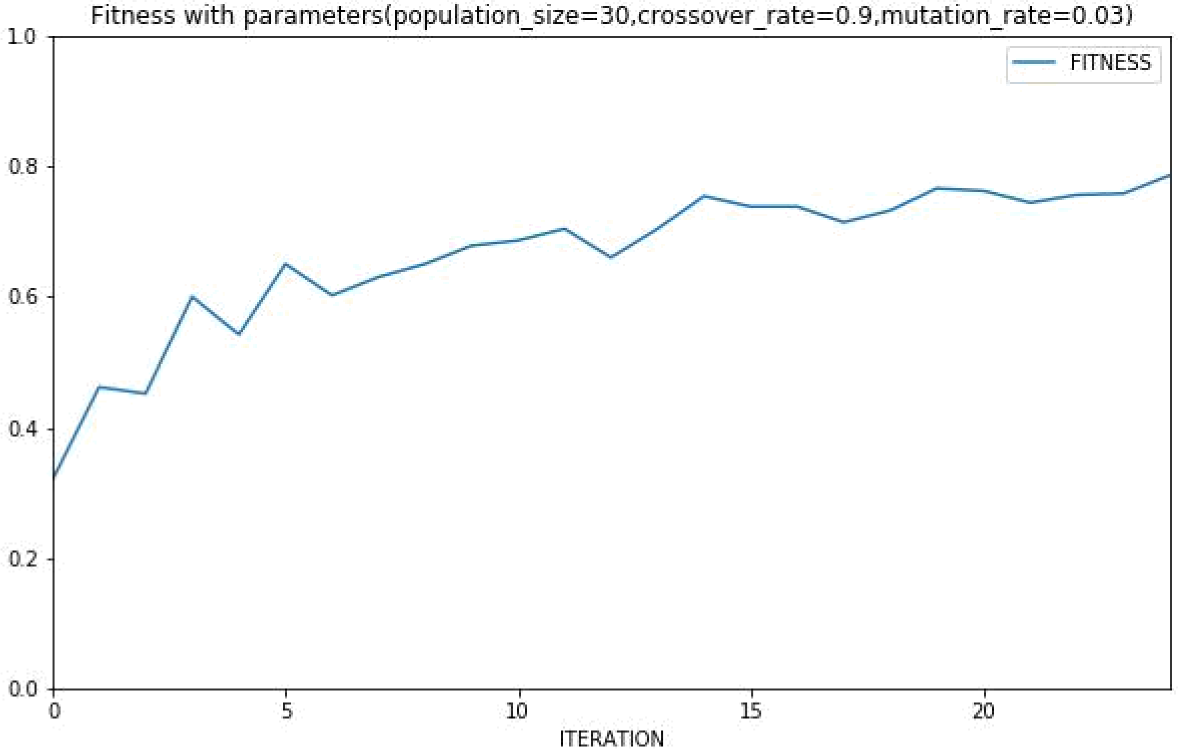

Fitness with parameters(population_size=30,crossover_rate=1.0,mutation_rate=0.03)
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
ITERATION
FITNESS

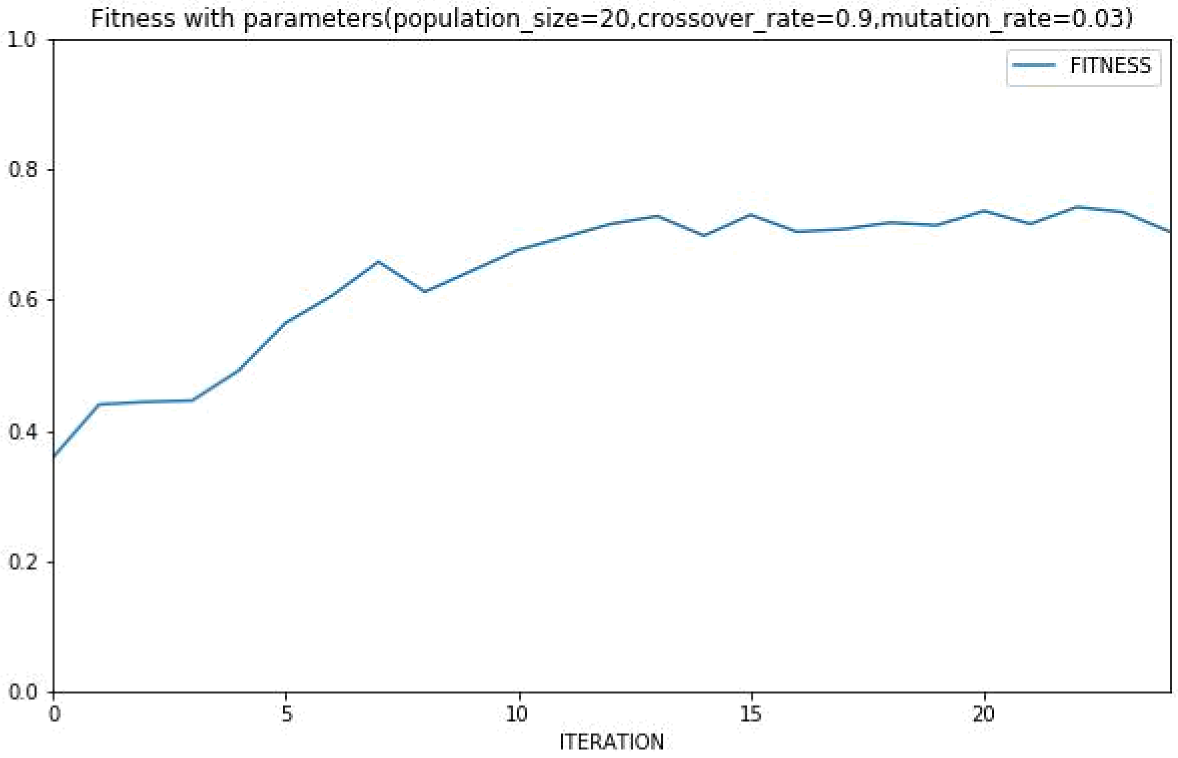


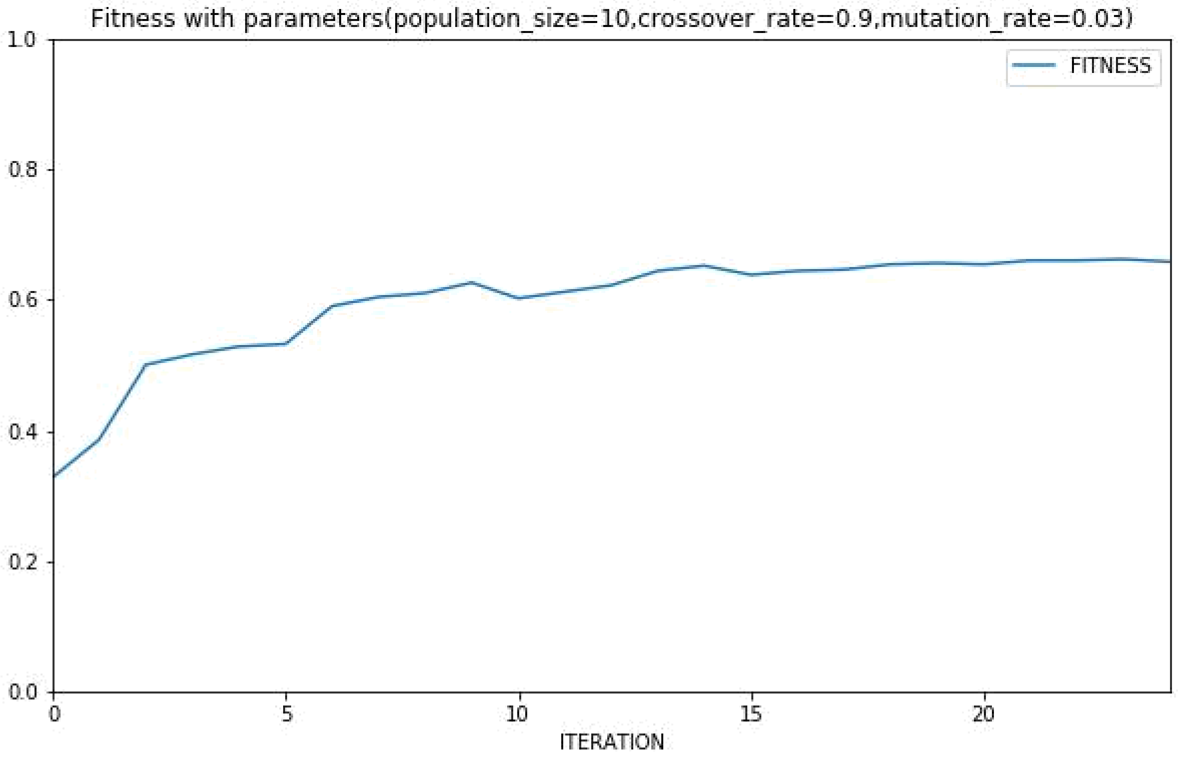
Fitness with parameters(population_size=20,crossover_rate=1.0,mutation_rate=0.03)
FITNESS
ITERATION
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20

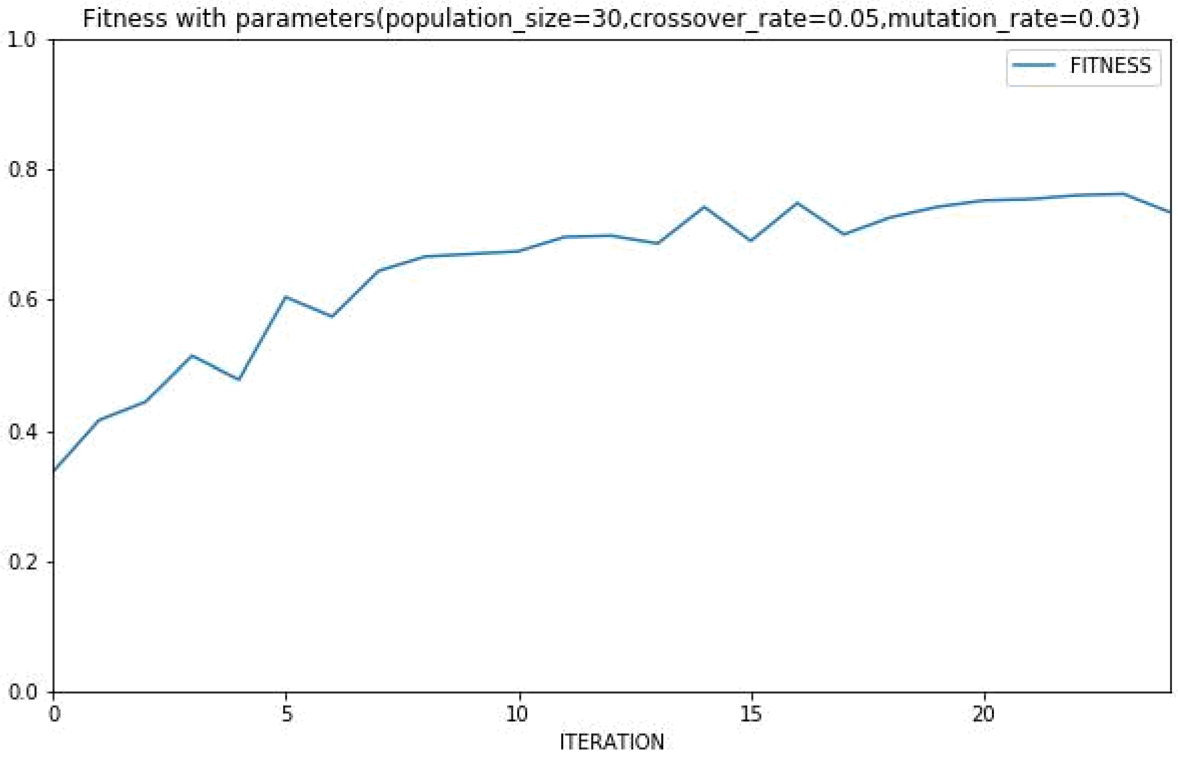


Fitness with parameters(population_size=10,crossover_rate=1.0,mutation_rate=0.03)
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
ITERATION
FITNESS

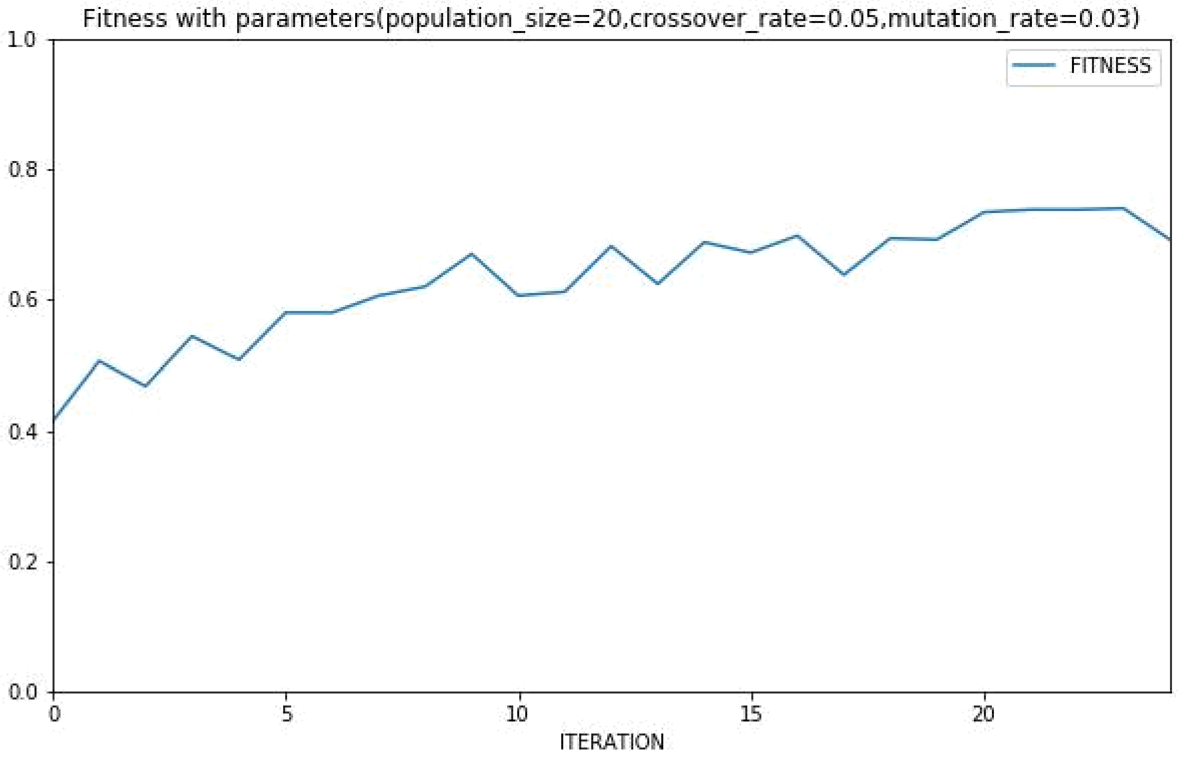

Fitness with parameters(population_size=30,crossover_rate=0.9,mutation_rate=0.03)
FITNESS
ITERATION
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20

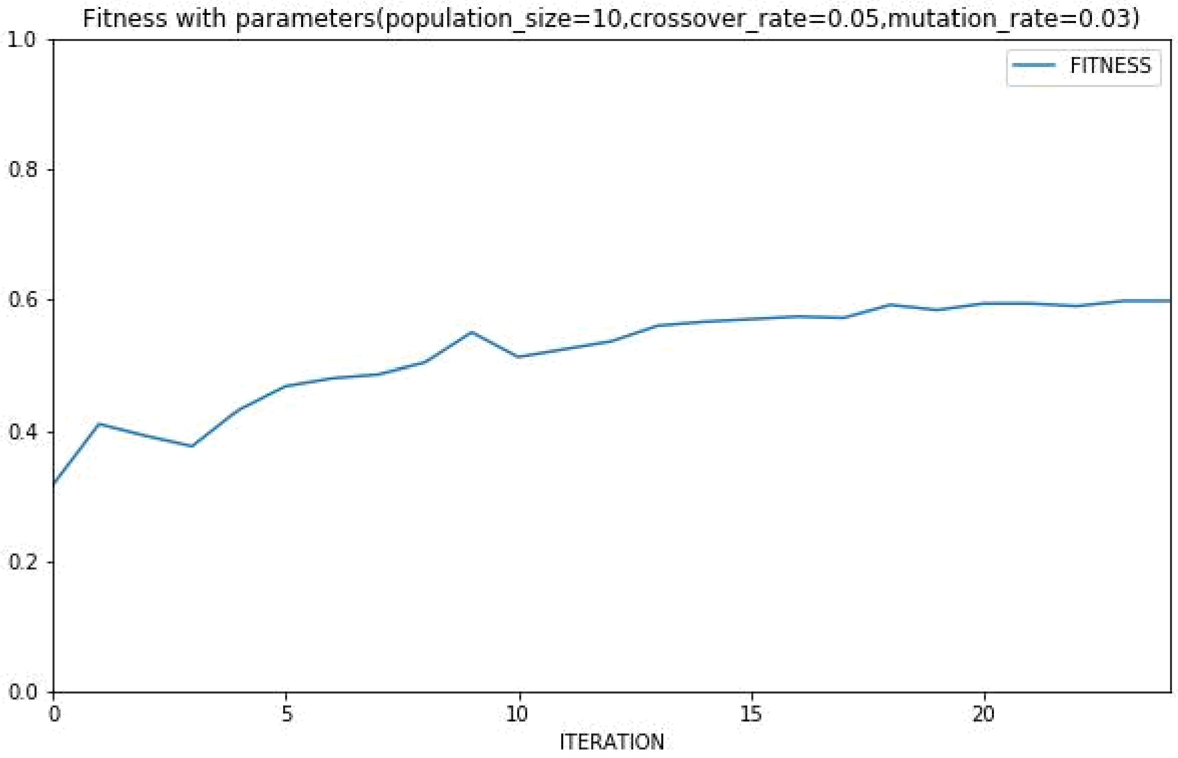

Fitness with parameters(population_size=20,crossover_rate=0.9,mutation_rate=0.03)
FITNESS
ITERATION
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20

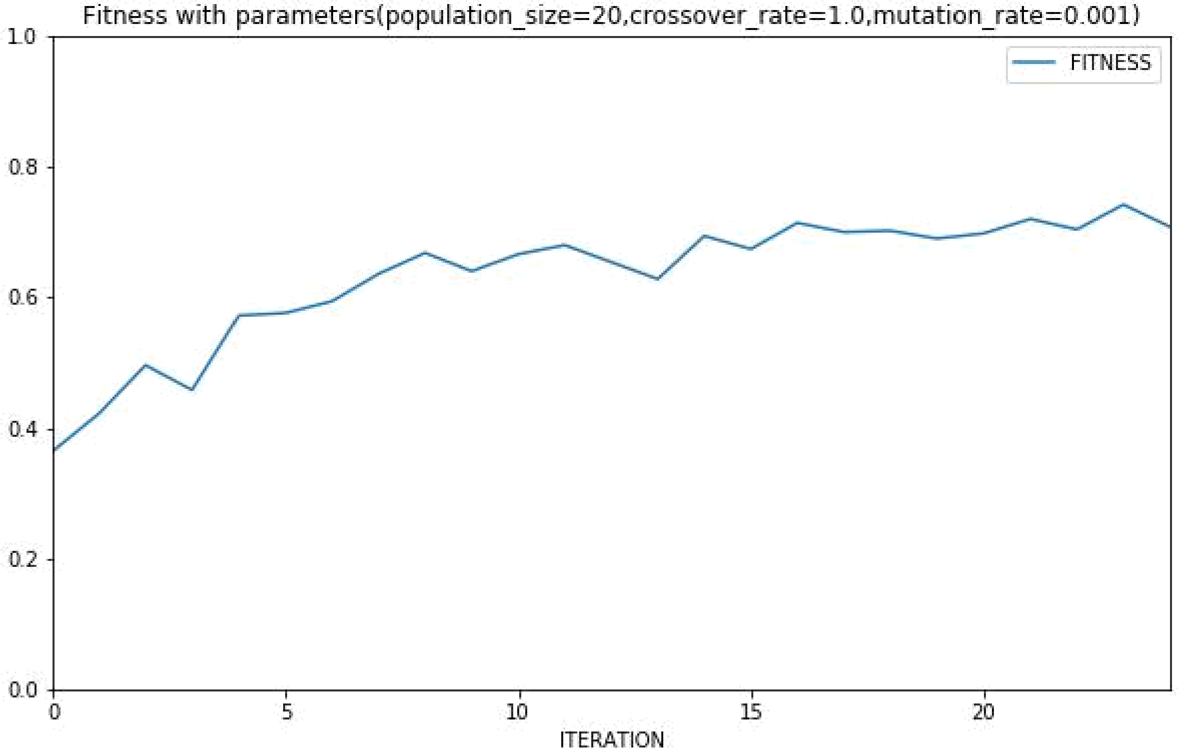

Fitness with parameters(population_size=10,crossover_rate=0.9,mutation_rate=0.03)
FITNESS
ITERATION
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20



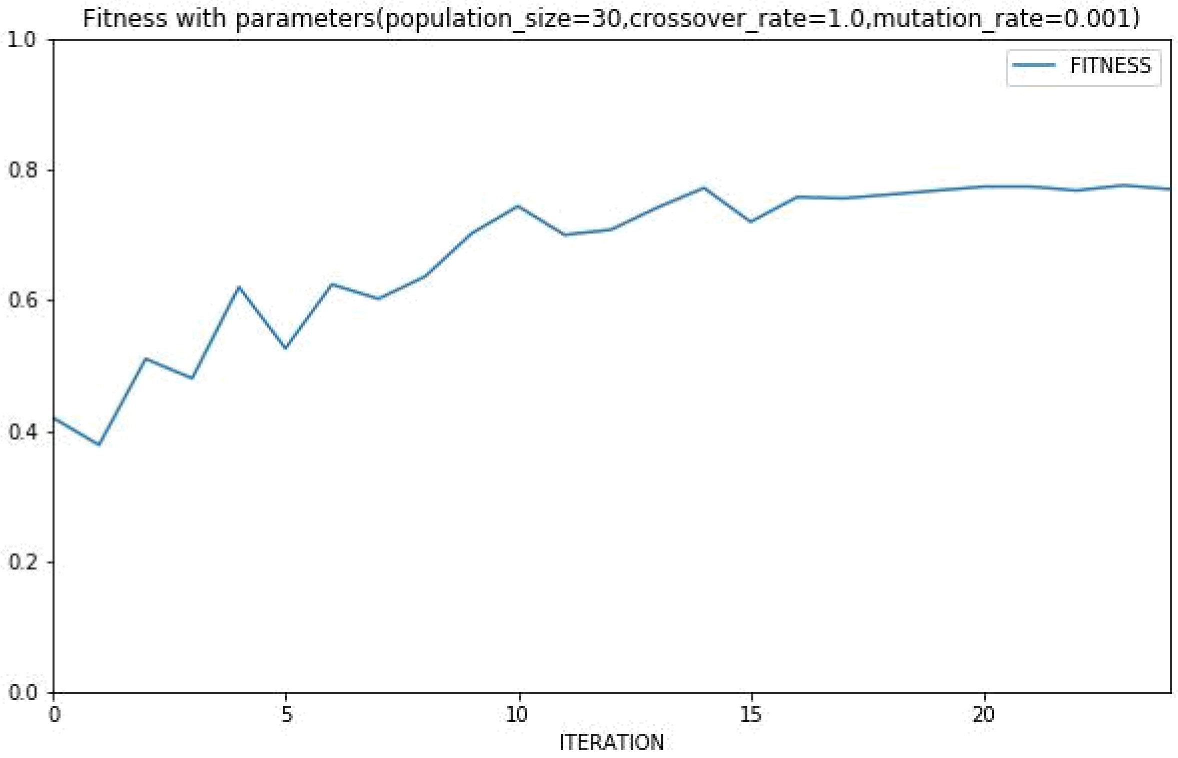
Fitness with parameters(population_size=30,crossover_rate=0.05,mutation_rate=0.03)
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
ITERATION
FITNESS


Fitness with parameters(population_size=20,crossover_rate=0.05,mutation_rate=0.03)
FITNESS
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
ITERATION


Fitness with parameters(population_size=10,crossover_rate=0.05,mutation_rate=0.03)
FITNESS
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
ITERATION



Fitness with parameters(population_size=20,crossover_rate=1.0,mutation_rate=0.001)
FITNESS
ITERATION
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20


Fitness with parameters(population_size=30,crossover_rate=1.0,mutation_rate=0.001)
FITNESS
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
ITERATION