SmartRehabilitation:

a GA-Based Solution for Optimal Exercise Plans

October - November, 2021

Group member: group# 9

Group members	ID
Lama Alamri	439201036
Ghada Altamimi	439200969
Ghaida Alomran	439200697
Sarah Aldrees	439200791

Supervised by:

T.Noura AlHammad

T.Nuha AlTayash

TABLE OF CONTENTS

SOLUTION REPRESENTATION

We use the Python language to solve the rehabilitation problem by using genetic algorithms (GA). We transfer the given table in the PDF file to a Comma-Separated Values (CSV) file, then we load the CSV file into the Python program to get the search space. #1

Index	Body Part	Exercise	Condition Type	Age Category
0	Elbow	Elbow extensor using free weights	Stroke	Adult
1	Elbow	Crawling	Stroke	Child
2	Elbow	Elbow flexor using free weights	Stroke	Adult
3	Elbow	Bear walking	Stroke	Child
4	Elbow	Elbow extensor using theraband	Spinal cord injuries	Adult
5	Elbow	Lifting in parallel bars	Spinal cord injuries	Child
6	Elbow	Elbow Flexor using theraband	Spinal cord injuries	Adult
7	Elbow	Lifting in sitting using scale	Spinal cord injuries	Child
8	Elbow	Rotating forearm	Brain injury	Adult
9	Elbow	Forearm supination	Brain injury	Child
10	Elbow	Learning forwards in a large ball	Brain injury	Adult
11	Elbow	Wheelbarrow walking on hands	Brain injury	Child
12	Upper Arm	Shoulder external rotator using free	Stroke	Adult
13	Upper Arm	Weight-bearing through one shoulde	Stroke	Child
14	Upper Arm	Shoulder extensor	Stroke	Adult
15	Upper Arm	Crawling	Stroke	Child
16	Upper Arm	Shoulder abductor using theraband	Spinal cord injuries	Adult
17	Upper Arm	Shoulder abductor Stritch in setting	Spinal cord injuries	Child
18	Upper Arm	Shoulder adductor using theraband	Spinal cord injuries	Adult
19	Upper Arm	Boxing in setting	Spinal cord injuries	Child

After that, the user will specify their optimal plan by entering age category (options include Adult and Child), condition type (options include Stroke, Spinal cord, and Brain injuries) and number of exercises (number of different exercises to be performed per body part). Then, we will represent (encode) each individual as an array of random indexes which contains the body part, exercise description, condition type, and age category for that exercise and it will be populated randomly with either 1 or 2 elbow exercises, 1 or 2 upper arm exercises, 1 or 2 knee/lower leg exercises, and 0 or 1 wrist exercises. Therefore, an individual can have a minimum of 3 exercises and a maximum of 7 exercises.

As shown in the below code:

```
def random_plan(klass, table, optimal_plan):
        exercises = []

        for i in range(random.randint(1, 2)):
            exercises.append(table.random_exercise(Exercise.ELBOW))

        for i in range(random.randint(1, 2)):
            exercises.append(table.random_exercise(Exercise.UPPER_ARM))

        for i in range(random.randint(1, 2)):
            exercises.append(table.random_exercise(Exercise.KNEE_LOWER_LEG))

        for i in range(random.randint(0, 1)):
            exercises.append(table.random_exercise(Exercise.WRIST))

        return klass(table, exercises)
```

For example, they may enter 1 elbow exercise, 2 upper arm exercises, 1 knee/lower leg exercise, and 1 wrist exercise. Therefore, the generated arrays will be with different length of random exercises.

Here is an example of 3 random encoded indexes:

[2 , 14 , 23 ,26, 38]

[8 , 19 , 26, 37]

[5 ,10, 14 ,20 30, 40]

Each index point to specific exercise as shown in below :

8	Elbow	Elbow Flexor using theraband	Spinal cord injuries	Adult
19	Upper Arm	Shoulder abductor Stritch in setting	Spinal cord injuries	Child
26	Knee/Lower leg	Knee flexor using a device	Stroke	Adult
37	Knee/Lower leg	Stepping sideways onto a block	Brain injury	Child

Each array of indexes represents one possible solution/individual.(each index is an exercise)

FITNESS FUNCTION

The fitness function uses the *Age Category*, the *Condition Type*, and the *Number of Exercises*, while the *Age Category* and *Number of Exercises* are equally important which means the *Age Category is 25%* and *Number of Exercises is 25%,* they are half as important as 3- the *Condition Type which has a 50%* so the they are all 100%.

$w1 = 0.25$, $w2 = 0.25$,
$w3 = 0.5$

where $\Sigma\, wi = 1$.

First, it calculates each of these as weighted sums and then creates a fitness from these weighted sums between 0.0 and 1.0.

For the *Age Category* and *Condition Type*, the weighted sum is computed by adding one when they match the optimal plan, and not adding anything (zero) if they do not match. For example, if the user inputted *Adult* and *Stroke* for the optimal plan, then this is the calculation:

```
age_category_sum = 0

condition_type_sum = 0
optimal_plan.age_catogery
>>Adult
optimal_plan.condition_type
>>Stroke

if exercise.age_category ==
optimal_plan.age_catogery: age_category_sum
+= 1

if exercise.condition_type ==
optimal_plan.condition_type: condition_type_sum
+= 1
```

Now, these values need to be normalized between 0.0 and 1.0, so they are divided by the number of optimal exercises:

number_of_exercises = 5

age_category_sum = age_category_sum / number_of_exercises
condition_type_sum = condition_type_sum / number_of_exercises

For the *Number of Exercises*, the weighted sum is computed by subtracting the optimal number of each exercise by what the individual has, and then it takes the absolute value of that for example

optimal_num_of_elbow= 1

optimal_num_of_upper_arm= 2

optimal_num_of_knee_lower_l
eg= 1 optimal_num_of_wrist= 1

The number of exercise generated by
GA

num_of_elbow= 1
num_of_upper_arm= 0
num_of_knee_lower_leg= 2

num_of_wrist= 1

The difference between individual and optimal exercise will be calculated using num_of_exercises_sum function

num_of_exercises_sum= (

abs(optimal_num_of_elbow — num_of_elbow) +
abs(optimal_num_of_upper_arm —
num_of_upper_arm) +
abs(optimal_num_of_knee_lower_leg —
num_of_knee_lower_leg) +
abs(optimal_num_of_wrist — num_of_wrist))

optimal plan has **2** upper arm exercises and the individual has **0**, then that's a difference of **2** meaning that the individual is off by **2,** so the sum of differences is

num_of_exercises_sum= 0+2+1+0=3

The problem with this sum is that lower values are good and higher values are bad. To fix this, the sum is subtracted from the maximum sum of possible value so, we need first to compute the maximum sum of possible value by multiply the number of body parts which is 4 by number of exercises, then we subtracted the num_of_exercises_sum(the difference between individual and optimal exercise) from maximum sum of possible value as shown in below :

max_sum_of_possible_value = 4 * number_of_exercises

no_of_exercises_sum = max_sum_of_possible_value —
num_of_exercises_sum

Now, this sum also needs to be between 0.0 and 1.0, so like the other sums, this number is divided by the maximum value:

no_of_exercises_sum = no_of_exercises_sum / max_sum

Finally, the requirements state that the *Age Category* and *Number of Exercises* should be equally important, but half as important as the *Condition Type*. which means

Therefore, they are multiplied by these percentages to compute the final fitness:

```
fitness = 0.0

fitness += 0.25 * age_category_sum
 # 25% fitness += 0.25 *
no_of_exercises_sum # 25% fitness
+= 0.50 * condition_type_sum
 # 50%
```

The final fitness is between 0.0 and 1.0, when it is close to 0.0 it means a worse fitness and when it is close to 1.0 it means a better. A fitness of 1.0 is the ideal, perfect goal and what the Genetic Algorithm is trying to achieve.

GENETIC OPERATORS:

CROSSOVER

As mentioned in the *Solution Representation* section, each individual is an array of exercises.

For the crossover, two parents are selected and a random crossover point (index) is selected depending on minimum length to avoid out of range exception.

```
num_of_exercises = min(len(self), len(partner))

crossover_point = random.randrange(1, num_of_exercises)
```

Then a child is created from the first parent and the second parent randomly, e.g the child array may be generated from 3 indexes from the first parent and 2 from the second parent.

For more explanation, here are two parents who are chosen by roulette wheel selection as described in the Selection by Roulette Wheel Selection..

First parent: [E11 U2 K4 W0]

Second parent: [E9 U6 K2 K11 W4]

Each *Exercise* is presented by using the first letter of a body part and a number of exercises in each body part.

first letter for body part ⇒ E for elbow, U for upper arm, K for knee/lower leg, and W for wrist.

Then the number is which exercise for that body part ⇒ *E11* stands for *E*lbow exercise number *11*.

To crossover, let's pick a random crossover point between 1 and minimum(length of first parent - second parent)

We begin from 1 because if we begin from 0, the child will match the second parent as well as, if the crossover point is the minimum(length of first parent - second parent), the child will match the first parent (when they have the same length).

So to prevent this, the crossover point is always between 1 and minimum(length of first parent - second parent) -1 , randomly to ensure a cross of both parents for the child.

```
num_of_exercises = min(len(self), len(partner))

#randrange() will be from 1 to num_of_exercises-1

crossover_point = random.randrange(1, num_of_exercises)
```

e.g. let crossover point to be index **1**.

First parent: [E11 U2 K4 W0]

Second parent: [E9 U6 K2 K11 W4]

0   1   2   3   4

Therefore, elements 0 from first parent will be taken and elements 1, 2,3, and 4 from second parent will be taken to form the new child as shown below:

Child: [E11 U6 K2 K11 W4]

The below code shows what we explained above:

```
 # cross from first parent

for i in range(crossover_point):
child_exercises.append(self._exercises[i])



 # cross from the second parent

for i in range(crossover_point, len(partner)):

      child_exercises.append(partner._exercises[i])



  # Create the child.

 child = self.__class__(self._table, child_exercises)

 return child
```

MUTATION

The munitions process begins with a random action either add, delete or replace an exercise.

```
randomAction = random.randint(1, 3)
```

A random mutation point (index) is selected between zero and the length of the optimal exercises.

```
mutation_point = random.randrange(0, length)
```

First, in the add exercise action:

The individual *Exercise* element may be added to increase the chance to reach the optimal plan but with one condition which is the size of the individual after the adding process must not exceed 7.

```
if randomAction == 1 and length < 7:

            # add one index.
mutated_exercises.append(self._table.random_exercise())
```

Second, in the delete exercise action:

The individual *Exercise* element may be deleted to increase the chance to reach the optimal plan but with one condition which is the size of the individual after the deleting process must not be less than 3.

```
if randomAction == 2 and length > 3:

                # remove one index.

                mutated_exercises.pop(mutation_point)
```

Third, in the replace exercise action:

one exercise element in the plan can be replaced with a new random exercise.

```
body_part = mutated_exercises[mutation_point].body_part

 mutated_exercises[mutation_point] = (

                self._table.random_exercise(body_part))
```

SELECTION BY ROULETTE WHEEL SELECTION

Selection for the entire population for each new generation, including crossover, is done by Roulette Wheel Selection.

In roulette wheel each Plan is given a “pie slice” (chance of being chosen) based on its fitness. If the fitness is bad, then the “pie slice” is smaller. If the fitness is good, then the “pie slice” is bigger. Therefore, the fitter individuals have a higher chance of being selected.

Before selection, the fitness for each individual in the population is computed, along with the total sum of the fitnesses of the population.

Next, an array of “pie slices” is built. Each “pie slice” corresponds to one individual in the population.

1-Creating an array of ‘pie slices’ (build roulette wheel slices)

```
    wheel_slices = []

    current_sum = 0.0

  for fitness in self._fitnesses:

            # Beginning of slice (range).

            wheel_slice = current_sum

            # Avoid dividing by 0 and negative (invalid) slices.

            if fitness > 0.0 and self._total_fitness > 0.0:

                wheel_slice += (fitness / self._total_fitness)

            wheel_slices.append(wheel_slice)

            current_sum = wheel_slice

        wheel_slices[-1] = 1.0

        return wheel_slices
```

2- Selecting the parents based on the roulette wheel selection

```
def select_index_by_roulette_wheel(wheel_slices, partner_index=-
1,selection_slice=None):

       length = len(wheel_slices)

       if length <= 1:

           return 0

       if selection_slice is None:

           selection_slice = random.random()

       if selection_slice <= wheel_slices[0]: # take the next index

           selection_index=0

       else:

         for i in range(0, length, 1):

           if i==length-1:

               selection_index=length-1

               break

           else:

               wheel_slice1 = wheel_slices[i]

               wheel_slice2 = wheel_slices[i+1]

               if  selection_slice >=  wheel_slice1 and selection_slice <=

                   wheel_slice2:

                   selection_index=i+1

                   break

           if selection_index == partner_index:

           if selection_index > 0:

               selection_index -= 1

           else :#the selection_index =0 take the next slice

               selection_index += 1  # 0 => 1

       return selection_index
```

Here is an example of all of these steps in action:

Fitnesses of all individuals in the population for example 10 individuals.

#we make an abbreviation for the names:

#E=elbow, U= upper arm, K= Knee/ Lower leg, W= wrist.

#St =Stroke, Sp=Spinal cord injuries, Br=Brain injury

#A=adult, C=Child.

index: [exercises] => fitness (0.0 - 1.0)

0: [EStA USpC UBrA KSpC WBrA] => 0.6000000000000001

1: [ESpA USpC UBrA KBrC WSpA] => 0.6000000000000001

2: [EStC USpC UStA KBrC WBrA] => 0.55

3: [EStA USpC UStC KStC WBrC] => 0.4

4: [EBrA USpA USpC KBrA WBrA] => 0.75

5: [ESpC UStA USpA KSpC WBrA] => 0.5

6: [EStC UBrC UStA KBrC WBrC] => 0.6

7: [ESpA UBrC USpA KStC WStC] => 0.44999999999999996

8: [EStA UBrC UBrC KSpC WSpC] => 0.5

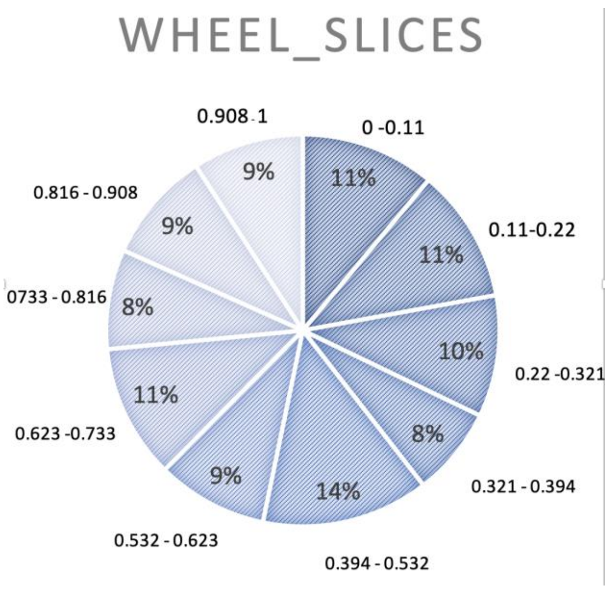
9: [ESpC USpC UStC KBrA WBrC] => 0.5

Array of wheel_slices.

index => end of pie slice range

```
0 => 0.11009174311926606          # 0.000 to 0.110 (11%)
1 => 0.22018348623853212          # 0.110 to 0.220 (11%)
2 => 0.3211009174311927 # 0.220 to 0.321 (10%)
3 => 0.3944954128440367 # 0.321 to 0.394 ( 8%)
4 => 0.5321100917431193 # 0.394 to 0.532 (14%)
5 => 0.6238532110091743 # 0.532 to 0.623 ( 9%)
6 => 0.7339449541284404 # 0.623 to 0.733 (11%)
7 => 0.8165137614678899 # 0.733 to 0.816 ( 8%)
8 => 0.9082568807339450  # 0.816 to 0.908 ( 9%)
9 => 1.0                          # 0.908 to 1.000 ( 9%)
                                    # Total sum: ~100%
```

This wheel that will generated from above example



```
# Rolling the roulette wheel.

# random number between 0.0 and 1.0 => index result

0.5270129358139273      =>   4
0.2575648493721574      =>   2
0.7986258407092812      =>   7
0.3638776517157697      =>   3
0.385169068123143       =>   3
0.9426967635169571      =>   9
0.24809478479196956     =>   2
0.5259533553114838      =>   4
0.11414043213317826     =>   1
0.5962637348074564      =>   5
0.1313283510486528      =>   1
0.7347172495979953      =>   7
0.36654262151991046     =>   3
0.7393003878497628      =>   7
0.46724860080138997     =>   4
0.03226299763334961     =>   0
0.18001017002995867   =>   1
0.04674362935969911     =>   0
0.17679752356038858     =>   1
0.026102524755209244    =>   0
```

REPLACEMENT

For simplicity, for each iteration, the entire population is completely replaced by selection for the next generation.

So, while the new population size is less than the target population size, an individual is selected by Roulette Wheel Selection.

We can only crossover if the random number generated is less than the crossover rate, also, we can mutate if the random number generated is less than the mutation rate.

This is included in the evolve function.

```
    def evolve(self):  # to create new population for the next generation by
crossover and mutation
            wheel_slices = self.build_roulette_wheel_slices()
             # New generation.
            new_population = []
        while len(new_population) < self._population_size:
                rehab_plan = None
                # Reproduce
             if random.random() < self.crossover_rate:
                 rehab_plan = self.crossover_by_roulette_wheel(wheel_slices)
             else:
                 rehab_plan = self.select_by_roulette_wheel(wheel_slices)
               # Mutate
              if random.random() < self.mutation_rate:
                    rehab_plan = rehab_plan.mutate()
```

Finally, it is added to the population.

```
            new_population.append(rehab_plan)


             # replace it with a new generation.

            self._population = new_population
```

TERMINATION CONDITION

Either one of the two conditions will terminate the Genetic Algorithm

First, it will terminate if an individual has a fitness of 1.0 which is the best fitness

Second, if the 1000th generation is reached, then it will terminate. From the experiments, this was found to be the optimal number of generations, as in most cases the perfect fitness was found before the 1000th generation, and it does not require a long wait by the user.

As well as, we did not use the error value as a termination condition, because we do not want to stop the process of searching for the best fitness which is equal to one unless we find it or we reach the 1000th generation.

RESULTS AND ANALYSIS

For all of the results, the following optimal plan was used:

Age category:	Adult
Condition Type:	Brain Injury
# of elbow exercises:	2
# of upper arm exercises:	2
# of knee/lower leg exercises:	2
# of wrist exercises:	1

The following Genetic Algorithm parameter settings were used:

Population sizes:	10	40	70
Crossover rates:	0.55 (55%)	0.75 (75%)	0.95 (95%)
Mutation rates:	0.05 (5%)	0.12 (12%)	0.20 (20%)

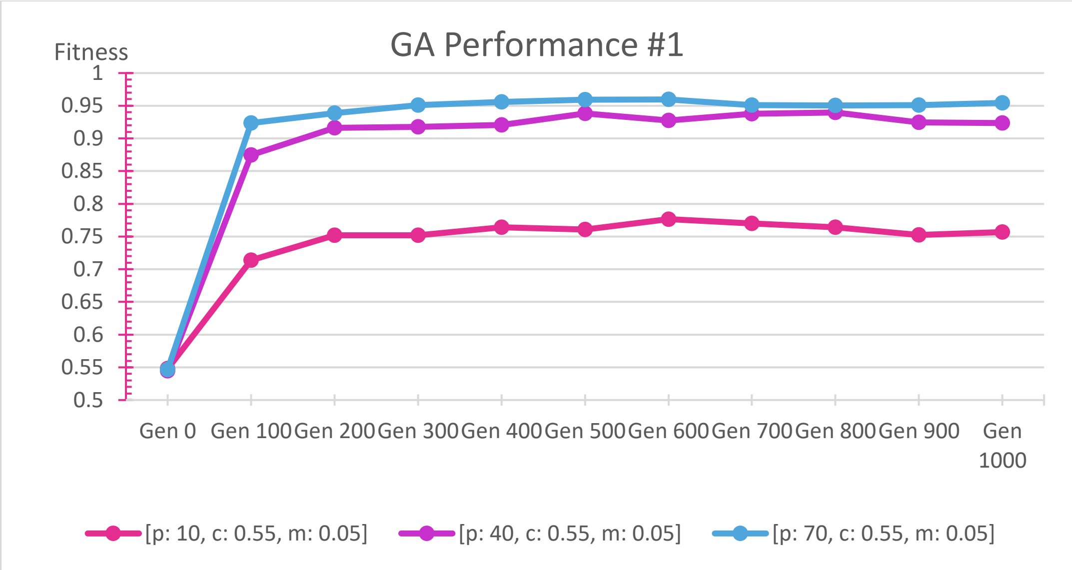
For each result, it was run 20 times. For each time, the Genetic Algorithm was initialized with random values and then evolved to the 1,000th generation. Finally, the average fitness was computed for each 100th generation .

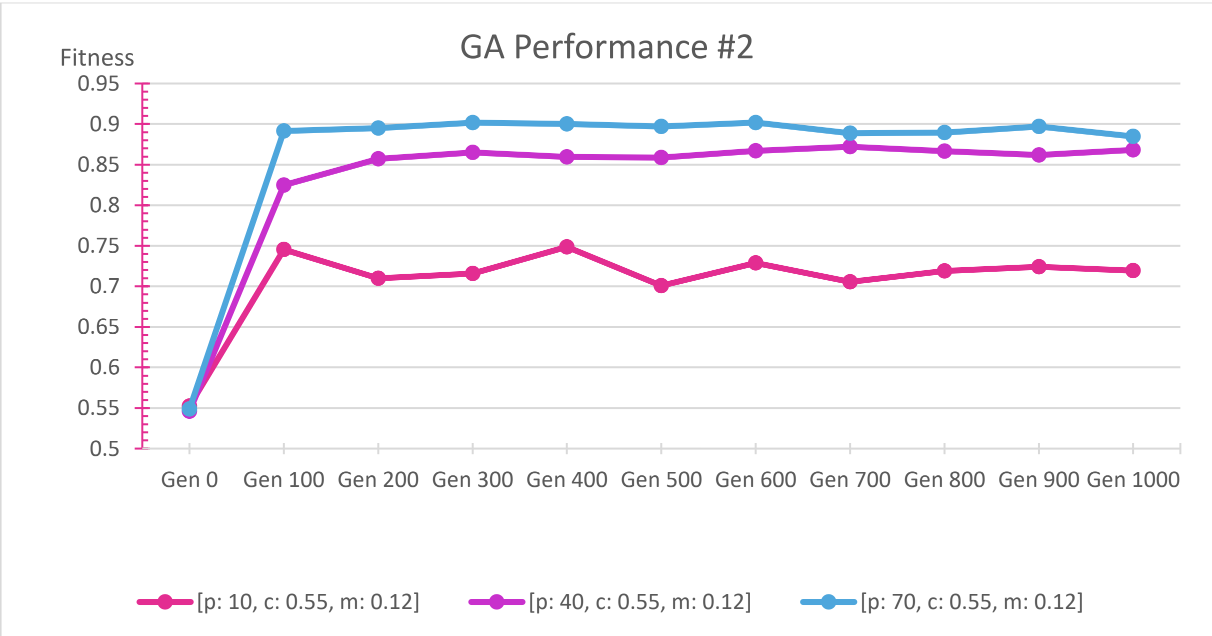
To save space on the charts, the following abbreviations are used in the name of the lines:

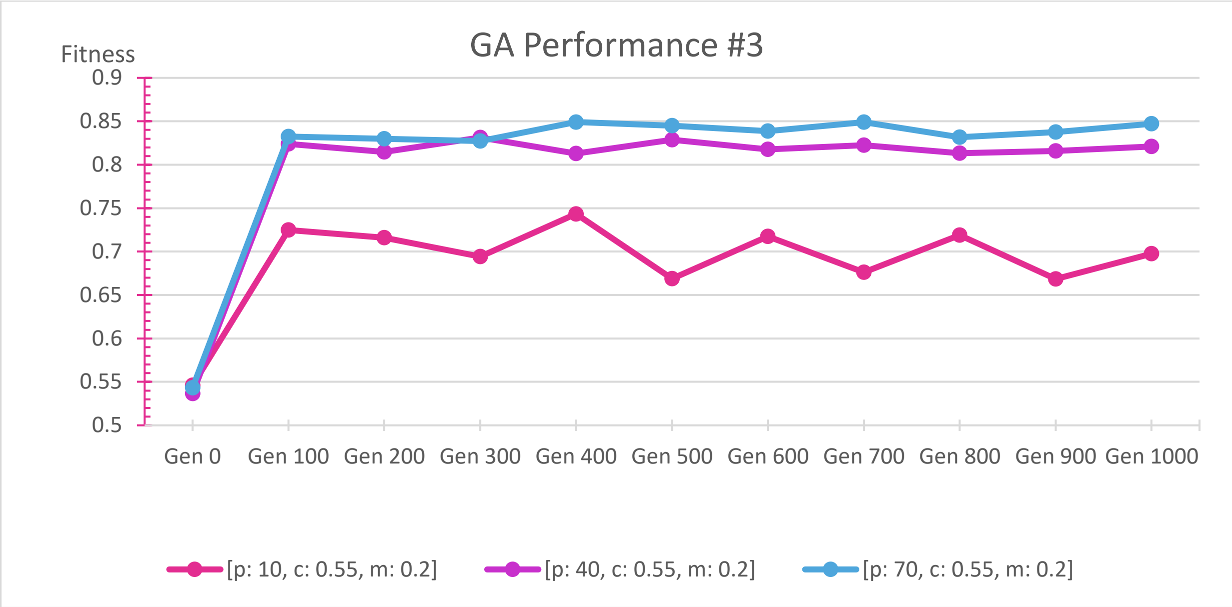
p: population size

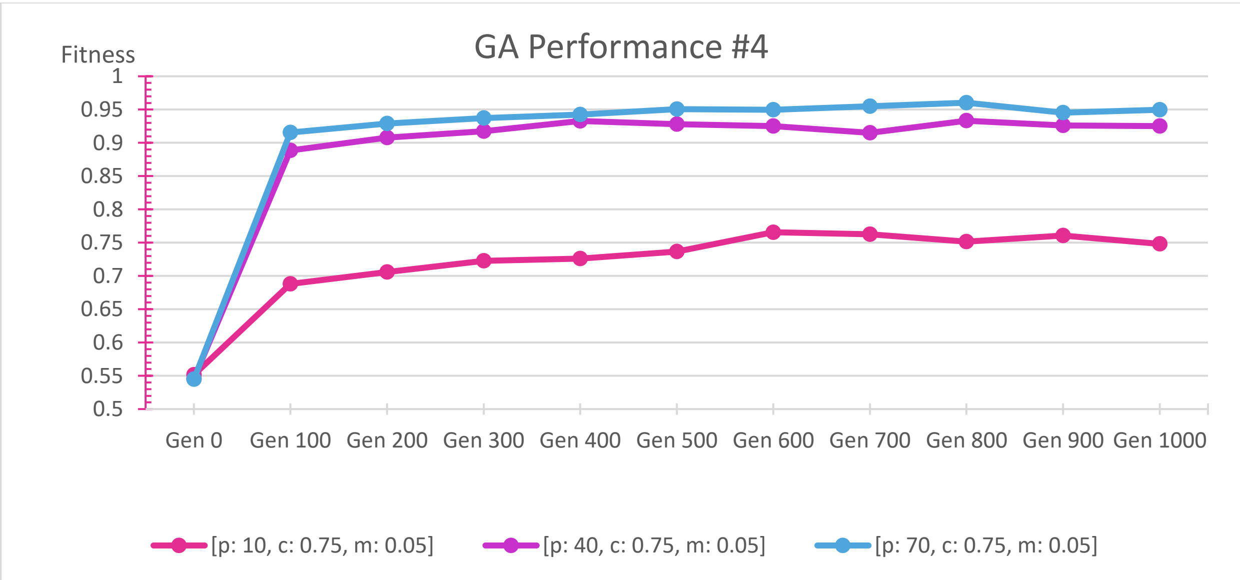
c: crossover rate

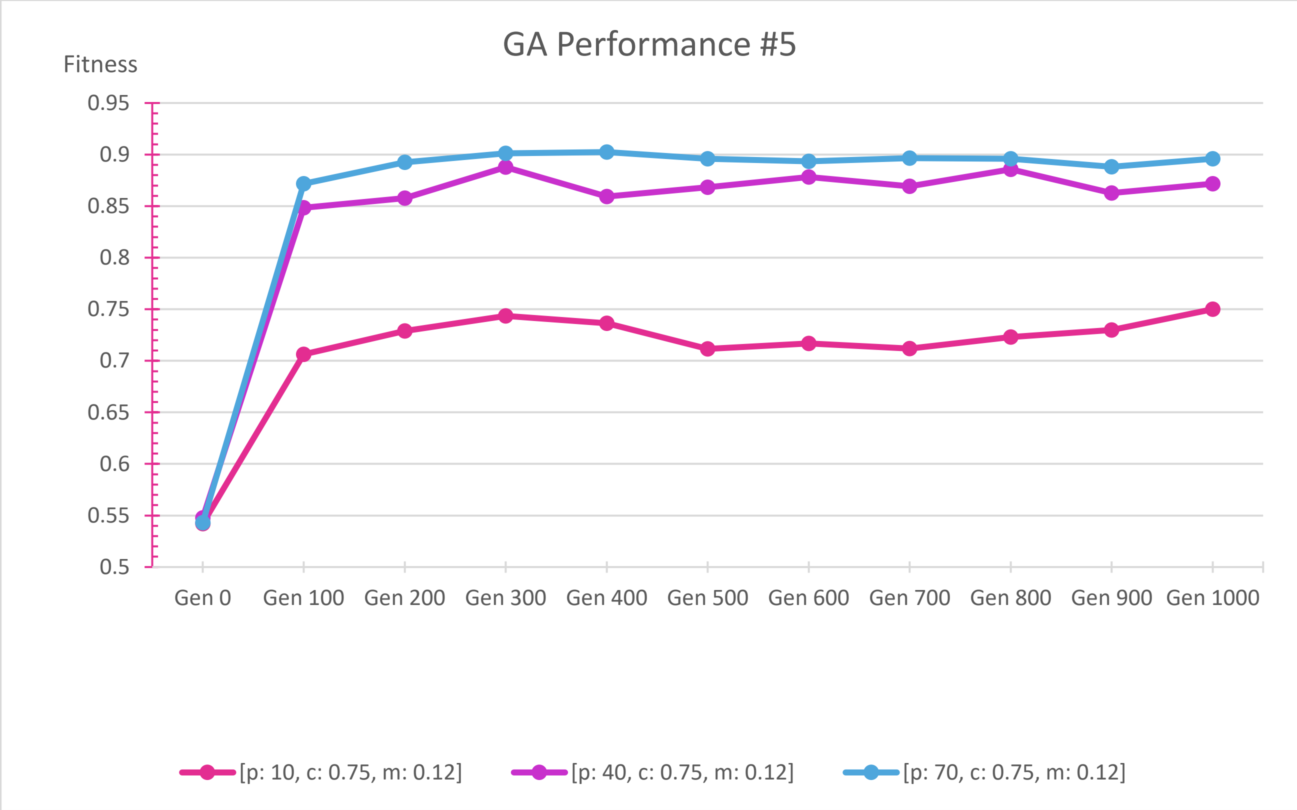
m: mutation rate

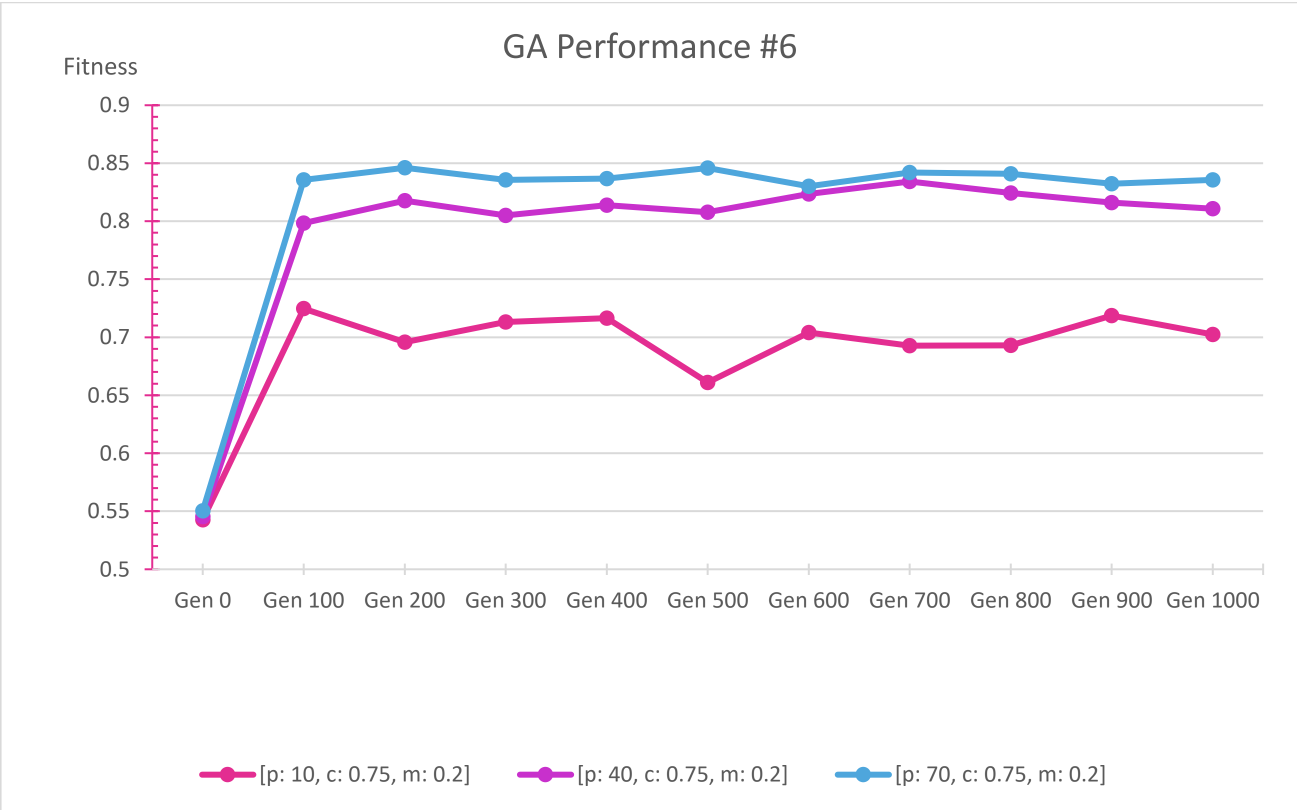

GA Performance #1
Fitness
1
0.95
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
Gen 0
Gen 100
Gen 200
Gen 300
Gen 400
Gen 500
Gen 600
Gen 700
Gen 800
Gen 900
Gen 1000
[p: 10, c: 0.55, m: 0.05]
[p: 40, c: 0.55, m: 0.05]
[p: 70, c: 0.55, m: 0.05]

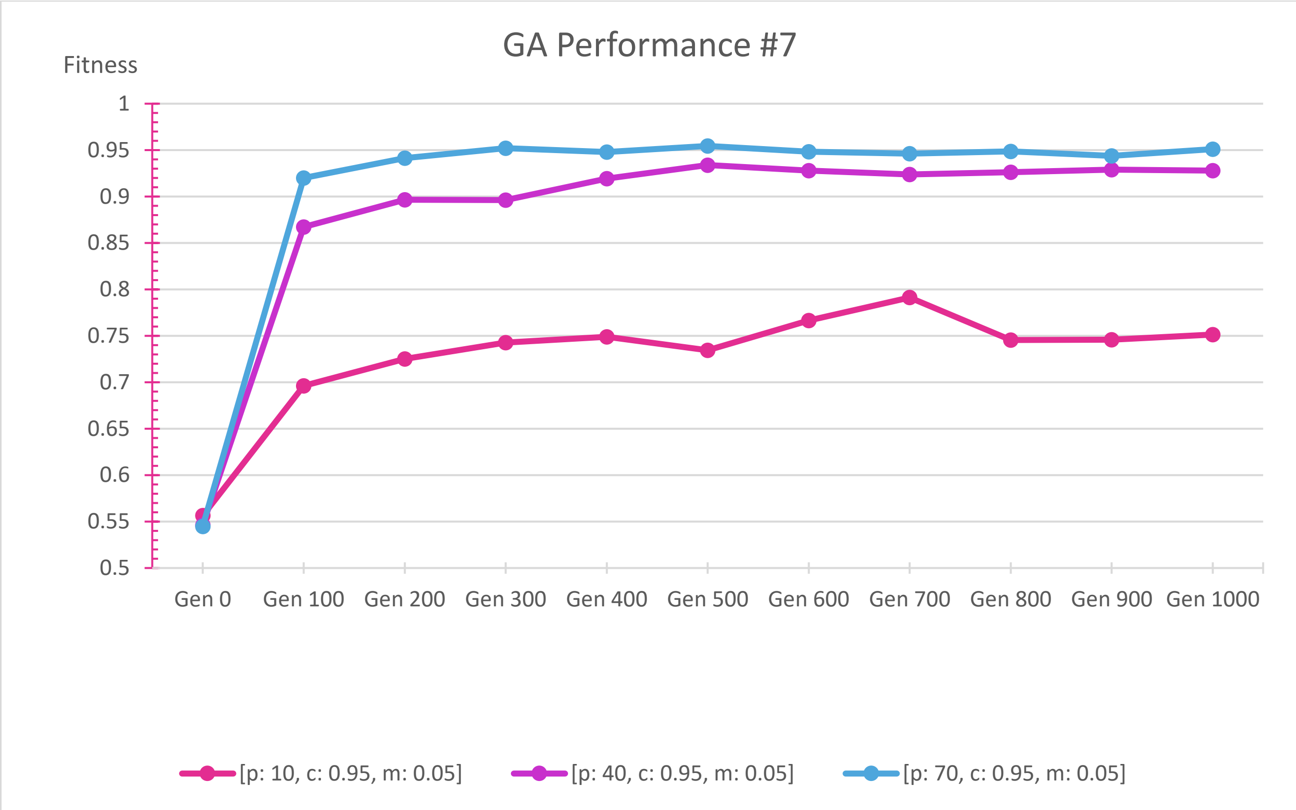

GA Performance #2
Fitness
0.95
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
Gen 0
Gen 100
Gen 200
Gen 300
Gen 400
Gen 500
Gen 600
Gen 700
Gen 800
Gen 900
Gen 1000
[p: 10, c: 0.55, m: 0.12]
[p: 40, c: 0.55, m: 0.12]
[p: 70, c: 0.55, m: 0.12]

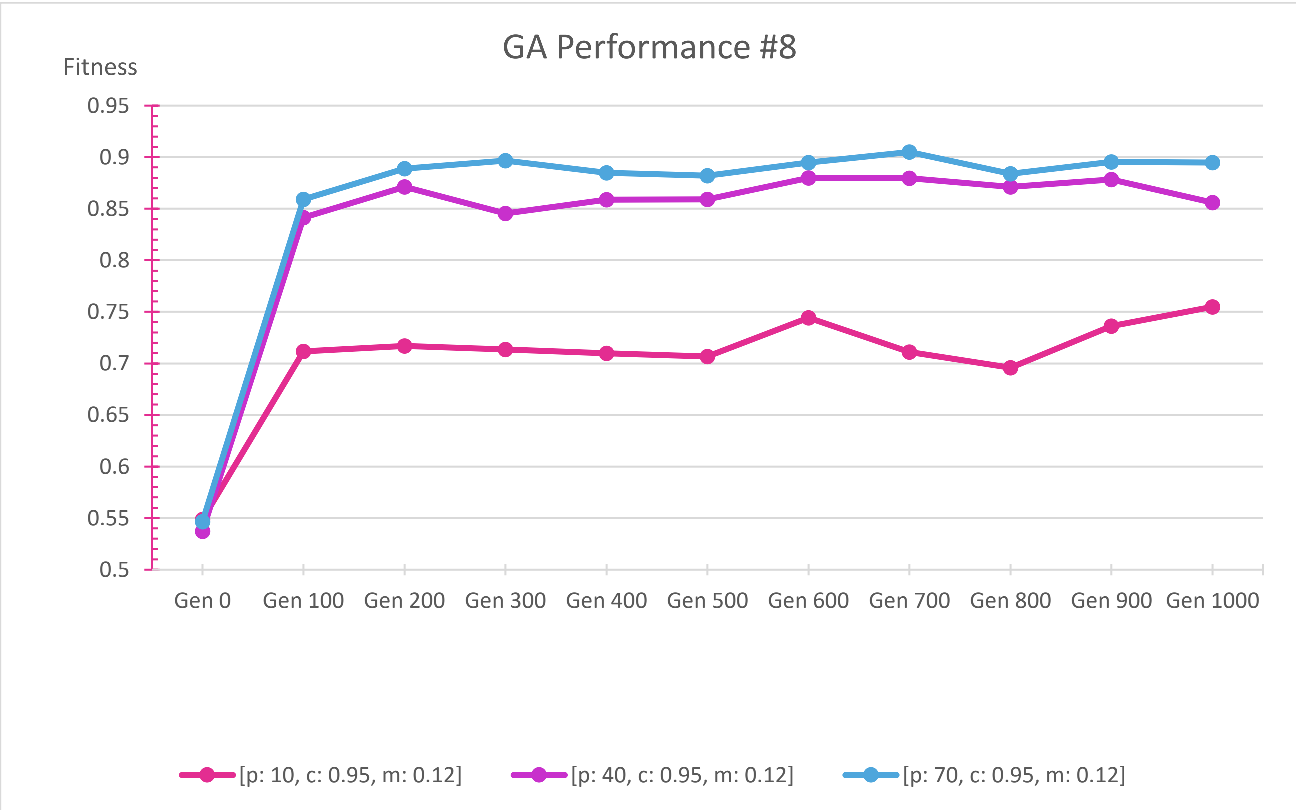

GA Performance #3
Fitness
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
Gen 0
Gen 100
Gen 200
Gen 300
Gen 400
Gen 500
Gen 600
Gen 700
Gen 800
Gen 900
Gen 1000
[p: 10, c: 0.55, m: 0.2]
[p: 40, c: 0.55, m: 0.2]
[p: 70, c: 0.55, m: 0.2]

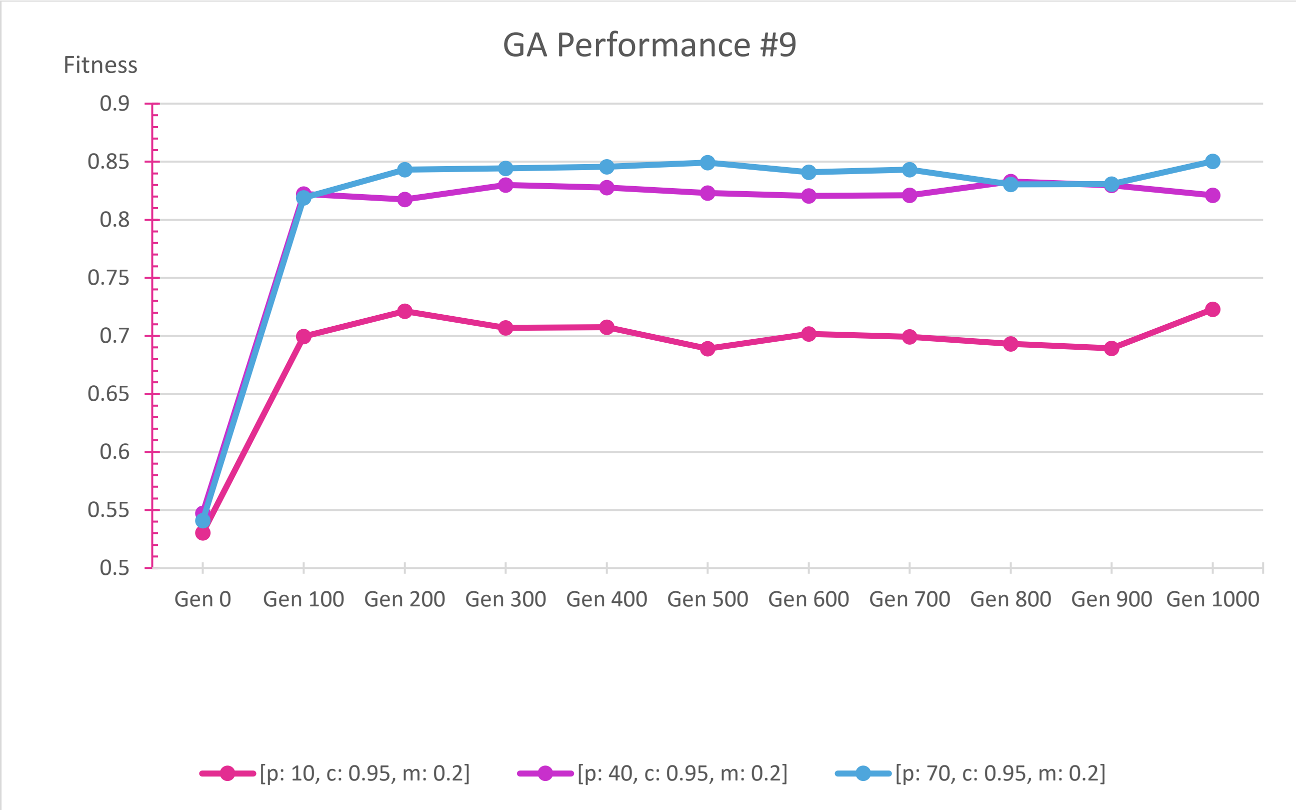

GA Performance #4
Fitness
1
0.95
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
Gen 0
Gen 100
Gen 200
Gen 300
Gen 400
Gen 500
Gen 600
Gen 700
Gen 800
Gen 900
Gen 1000
[p: 10, c: 0.75, m: 0.05]
[p: 40, c: 0.75, m: 0.05]
[p: 70, c: 0.75, m: 0.05]


GA Performance #5
Fitness
0.95
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
Gen 0
Gen 100
Gen 200
Gen 300
Gen 400
Gen 500
Gen 600
Gen 700
Gen 800
Gen 900
Gen 1000
[p: 10, c: 0.75, m: 0.12]
[p: 40, c: 0.75, m: 0.12]
[p: 70, c: 0.75, m: 0.12]


GA Performance #6
Fitness
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
Gen 0
Gen 100
Gen 200
Gen 300
Gen 400
Gen 500
Gen 600
Gen 700
Gen 800
Gen 900
Gen 1000
[p: 10, c: 0.75, m: 0.2]
[p: 40, c: 0.75, m: 0.2]
[p: 70, c: 0.75, m: 0.2]


GA Performance #7
Fitness
1
0.95
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
Gen 0
Gen 100
Gen 200
Gen 300
Gen 400
Gen 500
Gen 600
Gen 700
Gen 800
Gen 900
Gen 1000
[p: 10, c: 0.95, m: 0.05]
[p: 40, c: 0.95, m: 0.05]
[p: 70, c: 0.95, m: 0.05]


GA Performance #8
Fitness
0.95
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
Gen 0
Gen 100
Gen 200
Gen 300
Gen 400
Gen 500
Gen 600
Gen 700
Gen 800
Gen 900
Gen 1000
[p: 10, c: 0.95, m: 0.12]
[p: 40, c: 0.95, m: 0.12]
[p: 70, c: 0.95, m: 0.12]


GA Performance #9
Fitness
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
Gen 0
Gen 100
Gen 200
Gen 300
Gen 400
Gen 500
Gen 600
Gen 700
Gen 800
Gen 900
Gen 1000
[p: 10, c: 0.95, m: 0.2]
[p: 40, c: 0.95, m: 0.2]
[p: 70, c: 0.95, m: 0.2]

FINAL ANALYSIS

The previous graphs show that the higher population size, means better average fitness. Then we tried the 0.55, 0.75 and 0.95 as a crossover rate we noticed the difference between them was very simple and when the crossover rate was 0.95 it leads to a little bit better average fitness than 0.55 and 0.75. Lastly, in the above charts, it seemed that a higher mutation rate causes an overall lower average fitness. However, during testing, this actually is not a bad thing because it makes the population more diverse. The mutation rate of 0.20 actually made the Genetic Algorithm find the best solution (1.0) faster during tests. However, after 0.20, it starts to break apart and become too diverse and have too low of an average fitness.