Genetic Algorithm

Athavan Jesunesan

November 2022

1 Introduction

Genetic algorithms (GA) were introduced by John Holland in 1975 [Sim19]. GA is a type of evolutionary algorithm that can help to solve real world problems. Throughout this paper, I will be discussing how I implemented a GA in an attempt to help solve the recent development at We're Not a Scam Inc. where there is an abundance of shredded documents. In this situation I have implemented the GA using two different methods using: Ordered Crossover, and Uniform-Ordered Crossover to help unshred these documents. This implementation reveals some of the complexities of evolution when looking at the crossovers and mutation. Delving into this algorithm, how it could be optimized, and the results is worth noting regardless of its optimality. Researching and discussing these topics can help to broaden understanding and spark innovation.

2 Background

The following will be an overview of all the algorithms that compose this GA. This includes: the Main GA, Order Crossover, Uniform-Order Crossover, and Single-Point Mutation.

2.1 Main GA

The main GA orchestrates the rest of the algorithms to produce a GA. This is done by iterating through the following:

$i \leftarrow 0$
while $i \neq Maximum - Generations$ **do**
  if $i == 0$ **then**
    $Population[i] = NewPopulation()$
    $i++$
  else
    $Population[i] = Reproduce(Population[i-1])$
    $Population[i] = Crossover()$
    $Population[i] = Mutation()$
    $i++$

end if
end while

Using whichever crossover applies to the to algorithm. In this case, both the Uniform-Order Crossover and Order Crossover are used separately to compare and contrast results.

2.2 Order Crossover (OX)

The first crossover method used was Order Crossover. This implementation includes elitism. This means that the best 2 individuals in a given population will be automatically carried over to the next generation. For my implementation of OX, I took the 3rd individual and applied a crossover with the previous (starting at the 3rd since the first two cannot be removed or altered). The crossover takes a swath of 3rd individual's chromosomes and then fills in the rest of it using the previous individual. The crossover rate determines on average how many individuals will have there chromosomes crossed over. In the case where an individual lands in zone where they do not have to crossover they are added to the next generation. This was a deliberate choice to simulate randomness within the algorithm. The algorithm is as follows:

Algorithm 1 Order Crossover

$i \leftarrow Current - Generation$
$Population[i][1] \leftarrow FirstBestIndividual$
$Population[i][2] \leftarrow SecondBestIndividual$
$j \leftarrow 3rdIndividual$
while $j \neq LastIndividual$ **do**
  $rand = Random(100)$
  if $rand > CrossoverRate$ **then**
    $Population[i][j] = Population[i][j-1]$
  else
    $temp = swath(Population[i][j])$
    $Fill(temp, Population[i][j-1])$
    $Population[j] = temp$
    $j++$
  end if
end while

2.3 Uniform-Order Crossover (UOX)

Much like the OX, this crossover also includes elitism. For my implementation of UOX, I randomly created a mask which will keep approximately 40 percent of the locuses in the chromosome. The remaining 60 percent will be procedurally filled in using the prior individual. The mask is an array of 1's and 0's. When the mask has a 1 this means that the locus of the individual is kept. Otherwise, we fill in this position from the prior individual. The algorithm is as follows:

Algorithm 2 Uniform-Order Crossover

```
i ← Current − Generation
Population[i][1] ← FirstBestIndividual
Population[i][2] ← SecondBestIndividual
j ← 3rdIndividual
while j ≠ LastIndividual do
    if rand > CrossoverRate then
        Population[i][j] = Population[i][j − 1]
    else
        while k ≠ ChromosomeLength do
            Mask[] = CreateMask()
            if Mask[k] == 0 then
                Population[i][j][k] = Fill(Population[i][j − 1])
            end if
            k + +
        end while
    end if
    j + +
end while
```

2.4 Single-Point Mutation

In this experiment I have implemented a Single-Point Mutation. For this I generate a random point which represents a locus. I then generate another distinct locus. The last step is swap the two locuses at those points. The mutation is simple for the sake of not losing fitness of the chromosomes but implemented to ensure diversity. The algorithm is as follows:

Algorithm 3 Single-Point Mutation

```
i ← Current − Generation
Population[i][1] ← FirstBestIndividual
Population[i][2] ← SecondBestIndividual
j ← 3rdIndividual
while j ≠ LastIndividual do
    temp1 = RandomLocus()
    temp2 = DifferentRandomLocus()
    Swap(Population[i][j][temp1], Population[i][j][temp2])
    j + +
end while
```

3 Experimental Setup

To run a similar experiment to this ensure the following are implemented.

- Main GA
- OX
- UOX
- Single-Point Mutation
- Population is a 3D array where [] is the generation, [][] is the individual, and [][][] is a locus.

For my experiment I decided to create 10 generations for each crossover. Each generation was composed of 50 individuals. Each of these individuals has a chromosome length of 15. For the two crossovers that I used I included 5 different combinations of crossover and mutation rates. There were 5 different combinations that I used to apply to my GA. The first value represents the crossover rate and the second is the mutation rate: (100, 0), (100, 10), (90, 10), (90, 20), (80, 20). The total results I found for each document is (5 combinations)*(2 Crossovers)*(10 Generation) for a total of 100 runs.

4 Results

There are two main sections to to analyze here: Order Crossover, and Uniform-Order Crossover. The results are to be expected. The first values are the largest and as the generations progress, the fitness values slowly decrease. Some combinations of crossover and mutation rate worked better more consistently than others. There is plenty to look at with the results like; which combination was statistically best, which of the crossovers were best, which performed worst. The data is as follows:

We see for Figure 1 that most of the combinations plateaued around generation 8. The combination that seems to have performed best here is (80, 20), having the lowest fitness. This combination is still heading down and has not plateaued meaning that there was still room for improvement. I see in (100, 0) (Figure 2) that after generation 7 there was no more improvement. Comparing this to the graph we see that this combination seemed quite optimal at finding a good solution quickly. The combinations (100, 10), (90, 0), and (90, 10) (Figure 1) all seem to perform poorly. Each of these combinations plateaued early. (100, 0) has the best median and standard deviation.

In Figure 3, the combinations all seemed to starting plateauing near generation 6, about half way through the total generations. This UOX did not perform nearly as well as OX. The graphs show that even with the same combinations, the change of crossovers greatly impacted the quality of the solutions. The best combination here is (90, 10) (Figure 4). This combination shows some sign of still improving had it been given more generations. This is a big difference between UOX and OX as this combination was the worst for OX. Once again (100, 0) has plateaued early (Figure 4), however; it has the greatest standard deviation once again. We see that this combination performed well for both UOX and OX

Figure 1: Order Crossover (OX)

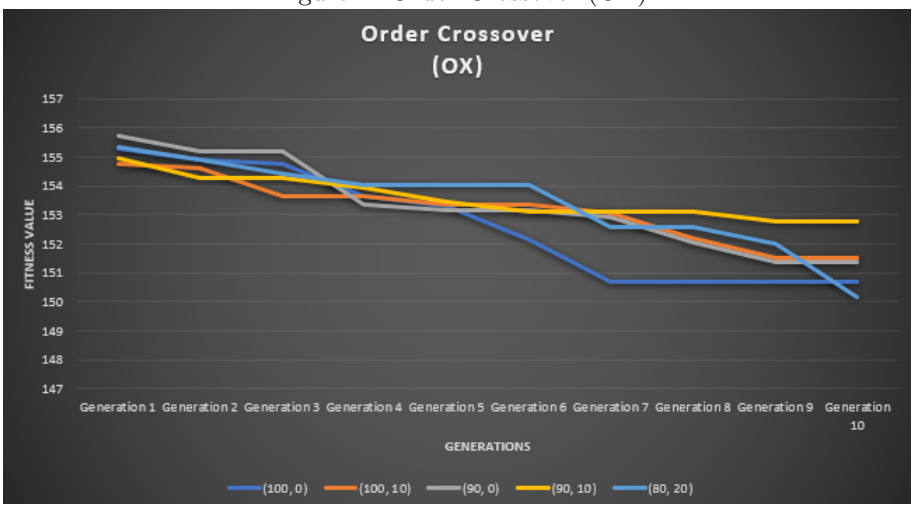

Figure 2: Order Crossover (OX)

	OX				
Mutation Rate	0	10	0	10	20
Crossover Rate	100	100	90	90	80
Generation 1	155.309224	154.7760216	155.7255205	154.9484049	155.3577173
Generation 2	154.8775159	154.6008522	155.2079972	154.2740833	154.8933712
Generation 3	154.7744431	153.6579585	155.2079972	154.264689	154.4194047
Generation 4	153.6266063	153.6579585	153.3490221	153.9241422	154.0365696
Generation 5	153.3106584	153.3587694	153.1709433	153.4548313	154.0365696
Generation 6	152.1447989	153.3587694	153.1709433	153.1041607	154.0365696
Generation 7	150.7029354	153.0757368	152.8938227	153.1041607	152.5648852
Generation 8	150.7029354	152.1663225	152.0313992	153.1041607	152.5648852
Generation 9	150.7029354	151.5359674	151.3841392	152.7545894	151.9805131
Generation 10	150.7029354	151.5359674	151.3841392	152.7545894	150.1774201
Average Fitness:	157.4318509	156.5514272	156.7223627	155.9582729	155.3961769
Min:	150.7029354	151.5359674	151.3841392	152.7545894	150.1774201
Max:	155.309224	154.7760216	155.7255205	154.9484049	155.3577173
Median:	152.7277287	153.3587694	153.1709433	153.279496	154.0365696
Standard Deviation:	1.826024517	1.073798773	1.492899332	0.706951147	1.487010416

Figure 3: Uniform-Order Crossover (UOX)

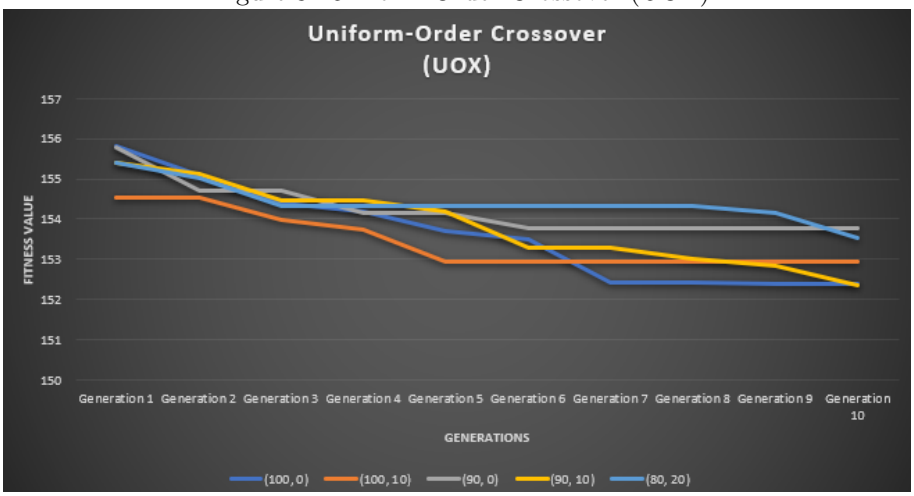

Figure 4: Uniform-Order Crossover (UOX)

	UOX				
Mutation Rate	0	10	0	10	20
Crossover Rate	100	100	90	90	80
Generation 1	155.8172403	154.5315119	155.7736018	155.4122498	155.4017237
Generation 2	155.1051928	154.5315119	154.697314	155.1284041	155.0322409
Generation 3	154.384684	153.9698165	154.697314	154.481984	154.3190601
Generation 4	154.1784209	153.7345379	154.1670062	154.481984	154.3190601
Generation 5	153.7131644	152.9485562	154.1670062	154.18291	154.3190601
Generation 6	153.4808409	152.9485562	153.7728723	153.2784079	154.3190601
Generation 7	152.4391774	152.9485562	153.7728723	153.2784079	154.3190601
Generation 8	152.4391774	152.9485562	153.7728723	152.9975533	154.3190601
Generation 9	152.3948126	152.9485562	153.7728723	152.8344864	154.1448695
Generation 10	152.3948126	152.9485562	153.7728723	152.3473471	153.5399335
Average Fitness:	157.4338458	156.9110607	156.834841	156.3494516	156.7904006
Min:	152.3948126	152.9485562	153.7728723	152.3473471	153.5399335
Max:	155.8172403	154.5315119	155.7736018	155.4122498	155.4017237
Median:	153.5970027	152.9485562	153.9699393	153.730659	154.3190601
Standard Deviation:	1.172474277	0.647997831	0.622237305	0.982513629	0.474044585
T Test	0.205904633	0.521335479	0.118403228	0.506336728	0.071456759

5 Discussions and Conclusions

Looking only at OX I see that, in terms of final fitness, the (80, 20) combination performed the best. We see that this one also has the most dramatic jump in fitness from generation 6 to generation 7(Figure 2). This was most likely a result of the 20 percent chance for mutation. With the large population of 50 and this mutation it is likely that some results would create spikes such as this one. This is the largest jump we will see in any of the data. It is also worth noting that this combination yielded the best average fitness. When looking at the graph it suggests that we did not hit the peak of this combination (Figure 1).It seems that given more generations the function would have continued to reach a solution. This combination for the most part is the best, however; it did not have the best median or standard deviation. The combination with the largest standard deviation would be (100, 0) (Figure 2). This shows that given 100 percent chance of OX that a good result can be found very quickly. Considering that the last 3 generations showed no improvement, had there only been 7 generations, the standard deviation would have been much more impressive. Looking again at Figure 2 we see that rest of the combinations all plateaued as well and the worst one being (90, 10)(Figure 1).

Based on these results the combination (100, 0) will come close to the solution the fastest, however, it is not a perfect solution. If speed was priority and the quality of the solution was not important then in this case, (100, 0) would be the best option. For OX (80, 20) seems to provide the most fit solution. Not only this, but given more time it would find the solution. This is due to the right mixture of exploration and exploitation. The downfall of (100, 0) is that it will plateau too quickly due to its lack of diversity. I believe that (80, 20) did not perform as well on UOX because of the nature of this crossover. This crossover already exchanges random portions of different individuals, and then fills them in. In essence this crossover is similar to the mutation algorithm(3). The issue may have been there was not enough exploration which is why the improvement of this combination did not perform as well as OX. Nonetheless, we see that this combination did not quite plateau and for UOX there is another big jump from generation 9 to generation 10 (Figure 4). This shows that in both cases had there been more generations this combination would most likely provide the best solution. Something worth noting is that the effectiveness of these combinations with the mutations, especially in UOX, is due to the elitism. Without this inclusion there is great risk that proceeding generations would have less fit fitness value than their ancestors. These results were also a product of how much diversity there was to begin with, having a population of 50 for each generation. This means that the starting elites have greater chances at being closer to the solution and that there are more odds that mutated chromosomes produce favorable fitness. In the end I see that for this problem using Order Crossover with a decent mixture of mutations and crossover, will yield a very fit solution, given it has a diverse starting population and enough iterations of generations.

References

[Sim19] David Simoncini. *Population-Based Sampling and Fragment-Based De Novo Protein Structure Prediction*. 2019. DOI: https://doi.org/10.1016/B978-0-12-809633-8.20507-4.