Document Recovery Using Genetic Algorithms

Shubham Amrelia
Brock University
April 03, 2023

I. Introduction

Genetic algorithms are the result of evolutionary biology used to make programs that work on an initial population. All the elements in this initial parent population are known as "chromosomes", and they each describe a possible solution to the given problem. To find how good of a solution a parent chromosome is, the method of calculating fitness is made. These parents are bred with each other to produce children which are expected to be a better solution to the problem i.e., to have a better 'fitness' value. But there are a lot factors that should be considered before selecting the parents for mating. There are ways of selecting parents such as k-tournament selection, fitness proportionate selection, roulette selection, etc. Then comes different ways of crossing the parent genes to create children, and whether or not the children will be mutated. Elitism is one such factor as well. It basically means to pass on a creatin amount of best parent chromosomes to the next generation automatically. This helps with obtaining convergence as the next generations, over the time, get many fitter chromosomes than the previous ones. All these factors will be discussed in detail in this report.

Shredders have been used for years to shred unwanted documents so that no person is able to recognize what the content of the document was. However, there is not a 'de-shredder' made yet that can accept strands of paper and combine into the exact document it was before shredding. The problem given to me during this assignment was to explore a genetic algorithm that can be beneficial for such a 'de-shredder'. For this assignment, we have been given with a digitally scrambled piece of document which we want to piece together using different permutations of an array. These different permutations are called 'chromosomes' and they each are a possible solution to the problem. Every chromosome will contain 15 elements ranging from 0 to 14. These elements are known as 'alleles'. We have been provided with a fitness function that accepts an array and calculates its 'fitness'. Fitness here implies how good of a solution the passed array is to the problem. The lower the fitness, the better. However, there can be millions of permutations of an array containing 15 elements. Instead of using all the chromosomes at one time, we will breed pools of chromosomes to produce children, while also passing the elite chromosomes automatically to the children. Furthermore, we will mutate the children to increase diversity and better our chances of finding the best chromosomes to use for deshredding. Over the course of multiple generation, we will notice that only the fittest of the population survives, hence enforcing the theory of evolution by Charles Darwin.

II. Background

Following is the sequence of pseudo code in which my GA works:

- Generates the initial population of n parent chromosomes, each containing 15 alleles
- Calculate the fitness values of the parents and record the indices of the elite parents
- For creating the children:
  - ⇨ Automatically include the elite parents
  - ⇨ Apply k-tournament selection on the remaining parents
  - ⇨ Apply crossover to two parents from the tournament selected parents until all the parents are 'crossover-ed'
  - ⇨ Finally, mutate the children using the mutation rate
  - ⇨ Return the children chromosomes and use them as parents for the next generation cycle

A. *Create the Initial population of 'popSize'*

First, we need to create a population of chromosomes that we can use to breed and create children chromosomes from. We do this by creating an Arraylist of integer arrays and filling it up with 'popSize' number of arrays that contain random number from 0 to 14 i.e., 15 elements. I made sure that there are no duplicates in a given array by using a Set, which I eventually convert into the ArrayList. We use 0 as random seed.

B. *Calculating Fitnesses and getting elite parents' indices*

Next thing I do is to calculate the fitness of each individual in our initial parent population and storing it into a double type array. Then I collect the elite parents by their relative fitness values. The size of elites is 8% of the popSize. I collect the indices of the elite parents for using them later on to pass to the next generation. A collection of indices is better as I need to compare the fitness values of the chromosomes significantly less number of times.

C. *K-tournament selection for the crossovers*

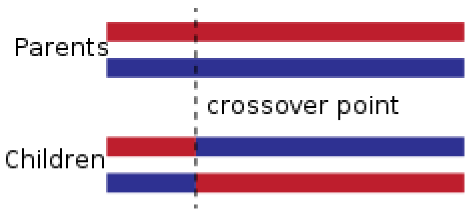
After that, I iterate through the parent population x times (x = popSize – eliteParents.size) and I choose 'k' parents randomly and choose the one that has the lowest fitness i.e., best fitness. This way I get a tournamentSelected array that contains all the winners is by collecting their indices to use them to retrieve those specific parents for the given population while doing crossovers. To do this, I choose k-random parents from the parent population and select the fittest parent chromosome index and add it to the tournament selected collection. I repeat this process for the size of population left after selecting eliteIndices.

D. Apply Crossover methods on elites and winner parents

The purpose of crossover function is to get two parents from the tournament selected population and alter their structure in a given way to imitate reproduction and produce children. The reason why we choose only the tournament selected parents is because the elite parent chromosomes are automatically added to the next generation. The number of parents selected for the crossover will depend on the crossover rate, which will be applied to 2 parents at a time. Having a crossover rate lets us retain some chromosomes that might improve our GA. In this GA, I have implemented two crossover methods that perform very different from each other. So, we select two parents from the tournament selected list and perform either of the following crossovers:

- Uniform Order Crossover (UOX)
  For implementation of Uniform Order Crossover, this method generates a bitmask with the same length as the parent chromosomes. We create 2 arrays named child1 and child2 of the same length. Here, the length is 15. The bitmask array will contain randomly generated values that can either be 0 or 1. Thereafter, we iterate through the bitmask and check the value at each index. If the bitmask value is 1, we pass the parent 1's allele at the current index to child 1. Similarly, while still in the iteration, we pass the parent 2's allele at the current index to child 2. After we are done iterating through the bitmask, we notice that there are some values missing in child 1 and child 2. This is because at the missing indices, the value of the bitmask was '0'. Hence, we iterate over the bitmask again and check if the value at the index is 0. If it is 0, we search through parent 2's indices and find their values. If the value at parent 2's current index is not present in child 1, then we pass it to child 1. We do this until child 1 is complete. We repeat the same process with parent 1 for child 2.

- Order Crossover (One – Point Crossover)



For implementing one point crossover, we need an index in the parent chromosome after which you can swap the elements and create the children. This index is also called crossover point and the result is two offsprings. The method I have designed checks if the child contains any of the elements that we need to swap with. For instance, let A be [1,2,3,4,5] and B be [4,6,9,3,8] and the swapIndex be 2. Therefore, child will be [1,2,3,8,4] and child2 will be [4,6,9,5,1].

E. Apply Mutation to our children at each generation

After applying the crossover, we get the next generation of children. These children have chromosomes that are altered from the parents, so they are expected to do a bit better when it comes to finding solution to the problem. Now that we have the list of children, we mutate them in a way that they introduce some randomness in the generation. The mutation rate is one of the parameters that user can change. It is calculated empirically for each child. We iterate through the children list and create a random value, if the value is less than the mutation rate, we mutate that child at that index. The way I have implemented mutation is by swapping the values two random indices of the child at the current index if it goes through mutation. For instance, child [1,4,3,7,6] becomes [1,7,3,4,6] (here we swap value at index 1 with index 3).

III. Experimental Setup

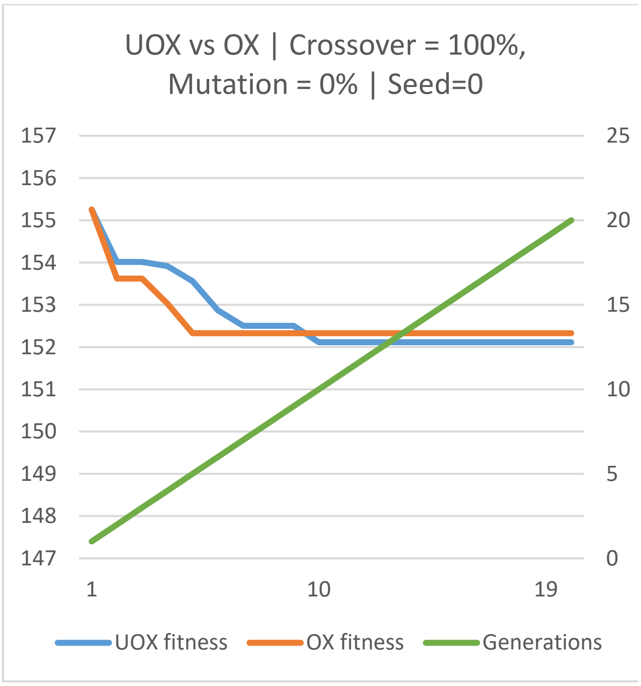
The experimental setup for my assignment contains a file thismyGA.java which runs with some preset parameters that the user can modify. My output runs for two crossover types and prints given parameters: random seed, crossover rate, mutation rate, crossover type. Next, it prints the best fitness value and the average fitness value for each generation including the initial population. Lastly it prints the best fitness solution by comparing the results of UOX and OX along with the solution chromosome. For the crossover rate, it accepts a double i.e. 90% = 0.9 and the same goes for mutation rate. Now, I have obtained the results for all the 5 cases that I was supposed to test. For the 5th parameter values, I chose a value that would give significantly differentiating results. These values are 90% crossover and 25% mutation.

My population size is set to 100 as it deemed sufficient for the GA, although it can be changed. Out of the 100 parents, my elitism rate is set to 8% so that the number of elite parents is always less than 10% of the population. The eliteSize can be changed but can only be an even number of percent of the population size for simplicity. For my k-value, I have chosen 5 as it would potentially increase competition between the randomly selected parents. I have set my generation span to 200 as I was able to obtain convergence within the 200 generations in each and every case I tested of different sets of parameters.

IV. Results

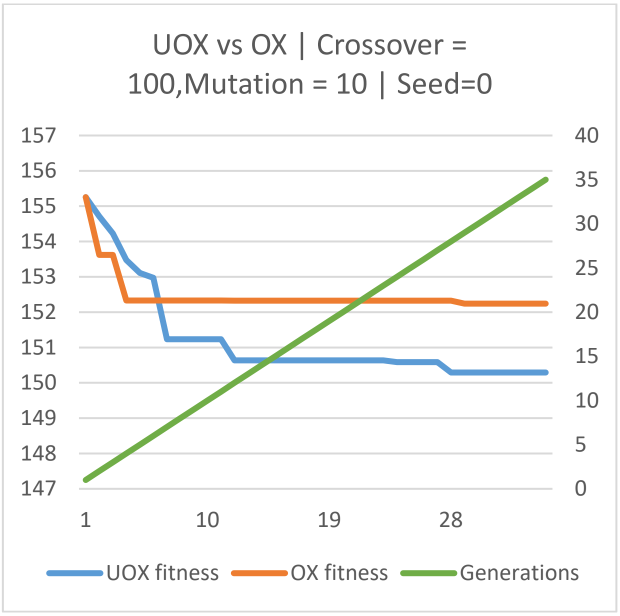
In this section I will list down my observations from the results of my GA using excel graphs. Please note that I am not talking the whole generation size but only to the point where it reaches convergence within 200. Note that all the observations I have noted down here are for document1-shredded.txt and using random seed 0. It shows the fundamental differences of using different parameters on each run.

A. Crossover Rate 100%, Mutation Rate 0%



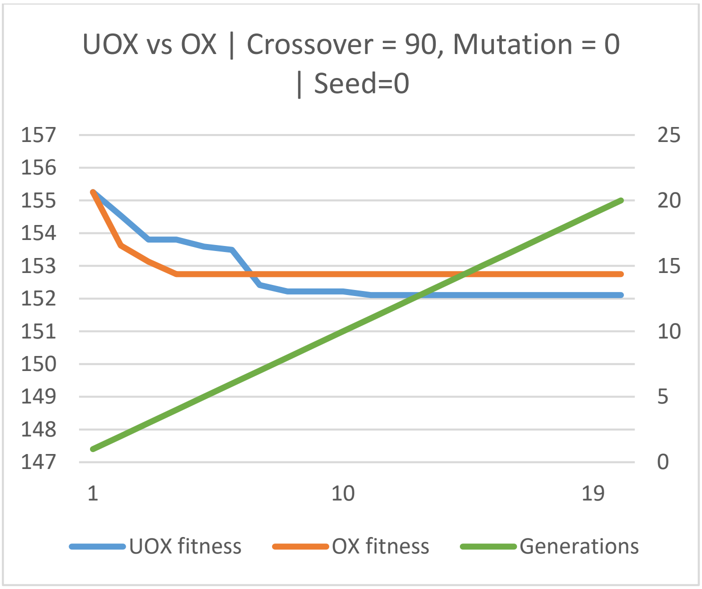
Comparing the Uniform Order Crossover with the One Point Crossover for this parameter set shows that while OX reaches convergence faster than UOX, the latter provides a better i.e., lower fitness value in the end over the span of generations. Moreover, there is more variance in the UOX graph the OX graph.

B. Crossover Rate 100%, Mutation Rate 10%



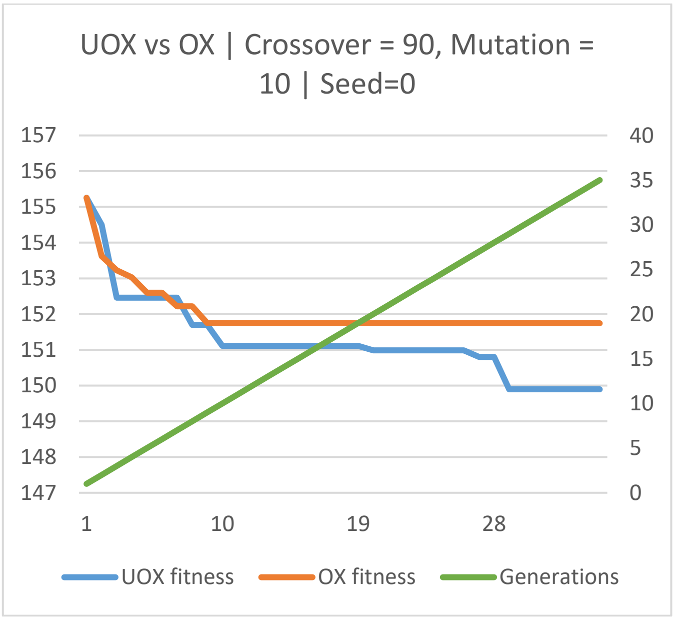
Comparing the Uniform Order Crossover with the One Point Crossover for this parameter set shows that there is a lot more variance in the fitness values by UOX, but Ox is not as diverse. Also, UOX reaches convergence faster than OX, while also providing a better fitness value. The highlight, however, is that when mutation is introduced in a controlled way (not too much, not too little), the best fitness value improves significantly. The graphs below will prove that as well.

C. Crossover Rate 90%, Mutation Rate 0%



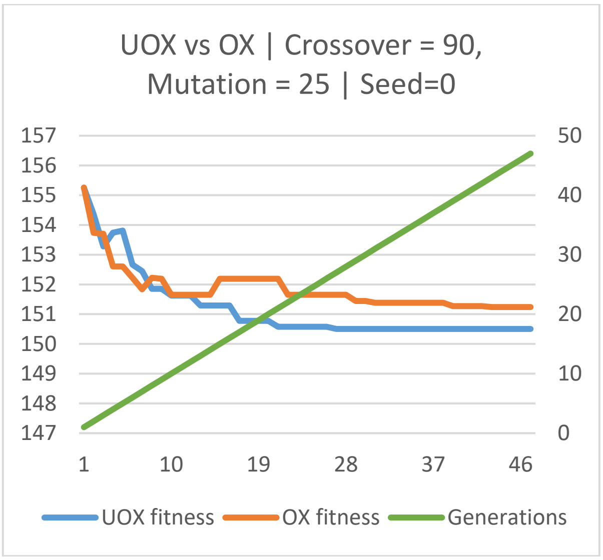
Comparing the Uniform Order Crossover with the One Point Crossover for this parameter set shows that while OX reaches convergence faster than UOX, the latter provides a better solution with a lower fitness value. Also, there is a lot more variance in the fitness values by UOX, but OX's results experience a steadier drop.

D. Crossover Rate 90%, Mutation Rate 10%



Comparing the Uniform Order Crossover with the One Point Crossover for this parameter set shows that OX reaches convergence much faster than UOX, but the latter provides a better solution with a lower fitness value. Also, there is a lot more variance in the fitness values by UOX, but OX's results experience a steadier drop. However, the thing that stands out is, we got a fitness value that is below 150 when we introduced a 10% mutation to the GA. This also shows that a crossover rate lower than 100% is better so that some of the average and some bad parents also pass on to the next generation to increase diversity. The result of them crossing with another chromosome could be significantly unexpected like getting a fitness value lower than 150.

E. *Crossover Rate 90%, Mutation Rate 25%*



Looking at the graph where there is a huge rate of mutation i.e., 1/4th of the children get mutated, we notice that the graph, instead of dropping either steadily or not, actually rises and drops multiple times. This shows that when there is lot of mutation in our GA, the best fitness values over the next generations are bound to get worse a lot more often before achieving convergence as compared to them getting better almost every generation when mutation is controlled (as mentioned previously, not too much, not too little).

F. *Statistical Analysis*

The statistical analysis that I have performed is by taking the best fitness and the worst fitness of the final population for 5 different parameters using random seed 0 and document1-shredded.txt.

The analysis shows that:

- The standard deviation for UOX is higher than that of OX crossover. In terms of statistics, this means that there is bigger variability in the data set of the best fitness found by UOX. I have emphasized this a lot when observing graphs too. Here we just get the proof of it.
- Notice that the average fitness (mean) for the UOX fitness over the span of several generations is lower than the mean OX fitness. This shows that UOX performs much better than OX.
- Whenever mutation is introduced to the GA, the final best population is better than when mutation is disabled (set to 0%)

	Crossover	Min	Max	Mean	Median	Std Dev
Cross 100%	UOX	152.1142311	155.2547917	152.1747857	152.1142	0.337054
Mutate 0%	OX	152.3281568	155.2547917	152.3592412	152.3282	0.247128
Cross 100%	UOX	150.2919704	155.2547917	150.4521097	150.292	0.655723
Mutate 10%	OX	152.2413113	155.2547917	152.2809132	152.2413	0.252645
Cross 90%	UOX	152.10887	155.2547917	152.1712009	152.1089	0.35361
Mutate 0%	OX	152.7480163	155.2547917	152.7668258	152.748	0.188714
Cross 90%	UOX	149.8958542	155.2547917	150.1357609	149.8959	0.715982
Mutate 10%	OX	151.7441935	155.2547917	151.7986272	151.7442	0.324183
Cross 90%	UOX	150.5040011	155.2547917	150.6678432	150.504	0.632145
Mutate 25%	OX	151.2395569	155.2547917	151.3824484	151.2396	0.446981

V. Discussions and Conclusions

The results that I have shed light upon in this report are obtained by using the two most important parameters: random seed 0 and document1-shredded.txt. Therefore, there are so many possibilities that my results could vary from when changing these crucial parameters to something else. My GA does allow the user to change these parameters and get the results desired for those parameters, however, it can differ from my analysis. The reason is that the shredded documents are different and hence they result in different fitness values, however, the lower is better. This could mean that OX might in some cases perform much better than UOX. However, the general observations I have made using my algorithm for the used parameters shows that it works as it is supposed to, by lowering the fitness values over the generation while also showing different results depending on the crossover rate, mutation rate and the crossover type as well.

User can also change the popSize variable to whatever they want, or the genSize which changes the number of generations to run the GA for. However, my results show that UOX gives consistently better fitness values as compared to OX crossover. This observation also spans when using mutation with both the crossover methods. I suspect this happens because One Point Crossover is steadier and does not experiment with randomness as much as Uniform Order Crossover. In OX crossover, the values in the arrays are swapped after a certain index for both the arrays. However, in UOX, we introduce a bitmask that is randomly generated so the certainty is more in the OX while the time randomness is used in UOX is more. Due to this, fitness values of OX drop steadily and has less variability as compared to UOX fitness values. This point is also proved by comparing the standard deviation as results of UOX have a larger deviation than those of OX (meaning more variability).

VI. References

https://en.wikipedia.org/wiki/File:OnePointCrossover.svg