Assignment Two: COSC 3P71 Genetic Algorithms

Ashu Chauhan 7001571
Department of Computer Science
Brock University
Niagara Falls, Ontario, Canada
ac20xp@brocku.ca

***Abstract*—**

***Index Terms*—genetic algorithms, optimization, crossover, mutation, selection, vigenère cipher, cryptanalysis**

I. Introduction

Genetic Algorithms (GAs) are a class of evolutionary algorithms that draw inspiration from biological evolution to solve complex optimization problems. They have been successfully applied to a wide range of domains, from engineering to economics, due to their robustness and adaptability. An example of a problem where GAs have shown promise is the Travelling Salesman Problem (TSP), which involves finding the shortest possible route that visits a set of cities and returns to the origin city.

In this study, we apply a GA to decipher a password used to encrypt text with the Vigenère Cipher, a simple yet historically significant cryptosystem. The effectiveness of the GA in this context hinges on various factors, including the crossover method, mutation rate, and selection process. We will explore these factors in depth, examining how each contributes to the algorithm's performance in cracking the cipher.

Furthermore, this report will provide a comprehensive analysis of the crossover methods employed, the selection techniques implemented, and the impact of mutation rates on the efficiency of the GA. To ensure repeatability of results, the study includes seeding details, enabling the replication of the experiments and verification of the findings from independent setups.

II. Background

Our program was developed in Java and consists of three classes. A Chromosone class which defines what a Chromosone is and generates a random key for each one based on a seed. A Evaluation class which was provided to us and developed by Tyler Crane. This class contains the code for the Vienère Cipher methods such as encrypting, decyrpting, and a basic fitness function. And lastly, a Main class which contains all the code to run and produce our results. This class contains the implementation of a GA through many methods containing the code for crossovers, mutations, reading data in from files, selection, generating populations and gathering elites since we used elitism for our implementation.

A. Application of Genetic Algorithms in Cryptanalysis

The application of GAs in cryptanalysis is based on the idea that the evolutionary process can be used to evolve decryption keys rather than having to use a brute force approach to search all possible keys for a solution. This approach is particularly useful in cases where the key size is too large and in turn makes finding a good solution much more efficient.

B. Chromosone Class

```
Fields:
    random: Random
    fitness: double
    key: array of char
    str: string = ”abcdefghijklmnopqrstuvwxyz-”
Constructor Chromosome(len: integer, random: Random)
    Initialize key with length len
    For each position in key, assign a random character from str
Constructor Chromosome(other: Chromosome)
    Clone the key from other
    Copy fitness from other
Method getFitness(): double
    Return fitness
Method setFitness(fitness: double)
    Set this fitness to the input fitness
Method getKey(): string
    Convert key array to string and return
```

We essentially assign each Chromosone with a key of length n, this is passed in as a parameter each time a new object is created, and along with this, a seed is provided to provide consistency in testing. The seed allows for repeatability so that you are able to achieve the same results each time if you pass in the same seed.

C. Main Class

```
Fields:
    gen: integer //generation number
    length: integer //key length
    elites: list of Chromosome //elites
    pop: list of Chromosome //population
    text: string //the string of text we are decrypting
    rndm: Random //seed
    alphabet: string = ”abcdefghijklmnopqrstuvwxyz-”
```

```
Constructor Main(seed: integer, crossoverRate: double, mutationRate: double)
    Initialize rndm with seed
    Call readFile method
    Call genInitPopulation method
    Call GeneticAlgorithm method with crossoverRate and mutationRate
Method readFile(filePath: string)
    Read file and store in text
Method GeneticAlgorithm(crossoverRate: double, mutationRate: double)
    Implement the genetic algorithm logic
Method tournamentSelection(): Chromosome
    Select and return a chromosome based on fitness
Method genInitPopulation(): list of Chromosome
    Generate initial population
Method getElites()
    Identify and store elite chromosomes
Method onePointCrossover(mother: Chromosome, father: Chromosome): Chromosome
    Perform one-point crossover and return new chromosome
Method uniformCrossover(parent1: Chromosome, parent2: Chromosome): Chromosome
    Perform uniform crossover and return new chromosome
Method mutate(individual: Chromosome, mutationRate: double)
    Mutate the given chromosome based on mutation rate
Method main(args: array of string)
    Main method to run the program
```

The Main class contains all the code which implements the GA. It calls upon the fitness method from the Evaluation class and takes in the parameters; seed, crossover rate, mutation rate. The class will get the seed number initially, then read a data file, generate an initial population of size 5000, and then run the GA. As part of the GA, there are several methods that were made for the GA as well such as the tournament selection method which will return a random chromosone which has a better fitness then the other from 2 randomly selected chromosones. Then there is a one point crossover method which implements the logic to perform a one point crossover. This is when a point is selected in the key and a child chromosone is produced from two parent ones that mix their key based on the crossover point. The get elites method will gather the two best chromosones from the population and store them for our new population. Uniform crossover method also implements the logic for this method. Mutate method will mutate an individual based on the rate at which is passed in as a parameter when the main method is called. The main method contains code which allows users to input the seed, crossover rate, mutation rate, and type of crossover and then runs the GA based on the given inputs so it is simpler to change the values for testing.

III. Experimental Setup

For experimentation, I ran the GA with the following values: a. Crossover rate = 100, mutation rate = 0 b. Crossover rate = 100, mutation rate = 10 c. Crossover rate = 90, mutation rate = 0 d. Crossover rate = 90, mutation rate = 10

If you decide to perform your own testing with my provided code, please adjust line 28 of the Main.java file and **REPLACE** DataX.txt with Data1.txt **OR** Data2.txt to swap between the data sets.

I performed each of the following tests with both Uniform and One Point crossovers and ran each test 5 times with 5 different seeds to see how results differed and how performance was impacted through each change.

IV. Results

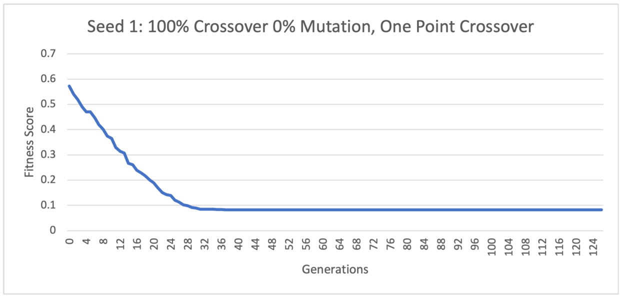

Fig. 1. Case A: Data 1: One Point

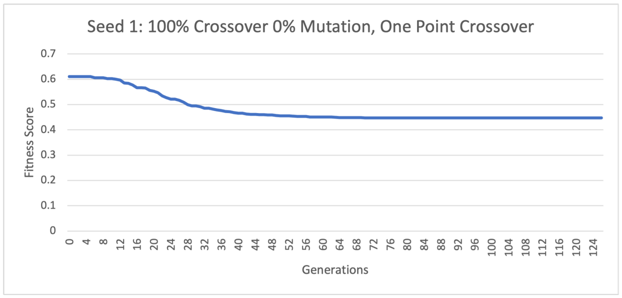

Fig. 2. Case A: Data 2: One Point

As seen through figure 1 and figure 2, each algorithm starts and then slowly or steeply dependant on the data source and key length comes down to a good solution, this same trend follows regardless of what seed we provided the algorithm with. The results did however improve when using Uniform Crossover and we arrived at the solution quicker as can be seen in figure 3 and figure 4.

These same trends followed for every seed tested and produced graphs that all looked very similar thus proving that regardless of the initial starting key, each algorithm reaches its optimal solution in n generations depending on key length and the text its decrypting if given the same parameters.

An adjustment of 10 percent in the mutation rate caused our results to decrease in quality severely. The effects of having a very high mutation right with a 100 percent crossover rate are

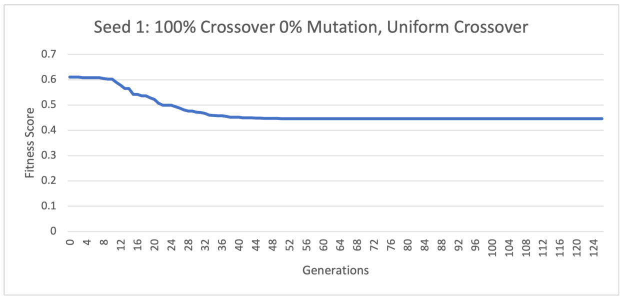

Fig. 3. Case A: Data 1: Uniform

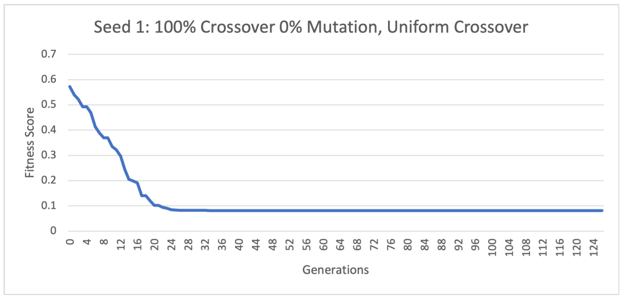

Fig. 4. Case A: Data 2: Uniform

catastrophic and lead to convergence in the algorithm. This can be seen through figure 5 and figure 6 as there are plenty of areas that look like stairs in the graph as that is where the algorithm struggled to move on from its current solution and find a better one, causing a plateau.

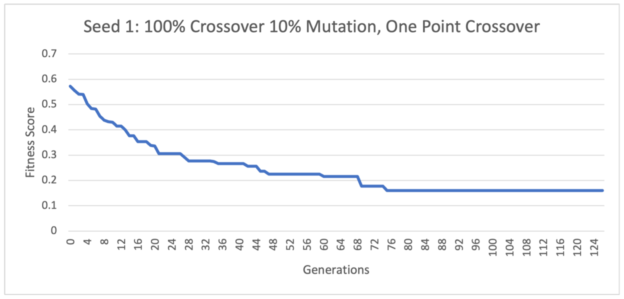

Fig. 5. Case B: Data 1: One Point

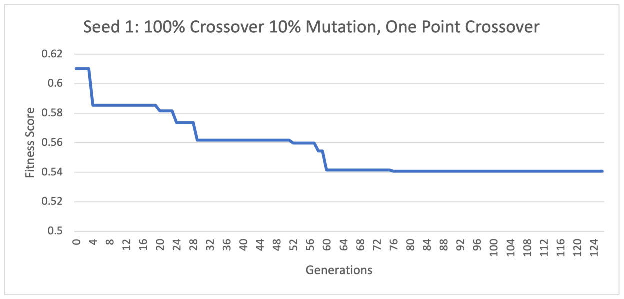

Fig. 6. Case B: Data 2: One Point

V. Discussions and Conclusions

Through analyzing the different effects that each parameter has on the performance of the algorithm, we have gained a better understanding of Genetic Algorithms as a whole. GAs mimic natural evolution, and its clear that if too much mutation occurs, this can be harmful. If there is too much crossover between two parents, this increases diversity, if there is not enough, this could cause bias in the population. The crossover method also has an impact on the performance as not all of them are the same or equal. All in all, my findings have allowed for a deeper understanding of the impacts your parameters have on the performance of a GA. It is important to optimize your values to ensure you get the best results possible in as little computing power as you can.

References

[1] "Vigenère cipher," https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher, accessed: November 10, 2023.

[1]