Prediction of Content Using Genetic Algorithm

Dieudonne N. Ouedraogo

SSIE 519 FINAL PROJECT

Prof. Harold Lewis

Department of System Science and Industrial Engineering
The Thomas Watson School of Engineering and Applied Sciences
The State University of New York at Binghamton, NY 13905.

ABSTRACT

The goal of this paper is to implement a genetic algorithm using python programming language to predict texts initially entered by a user. The program could be extended and used to retrieve user passwords, but the goal here is not to unlock passwords but rather to return content initially entered by a user. The program is written in python 3.5 an open source software. No special package other than the random library is used in this implementation, which allow us to simulated randomness by using pseudo-random numbers. The program is designed to be an interactive tool to gain good understanding of the importance of the parameters in GA. The program will prompt a user to enter a text and parameters values, the program uses Genetic Algorithm to predict the initial text entered. Depending on the parameters of the Genetic Algorithm: mutation rate, crossover rate, size of text, population size and maximum number of generation allowed, the program could succeed or fail in predicting the content. Attempt to measure the fitness through different generations will be addressed.

Keywords: Genetic Algorithm, Mutation, Crossover, Python

1. INTRODUCTION

Genetic Algorithms are a biological inspired stochastic metaheuristic for a combinatorial search and optimization. It is a strategy that mimic natural selection and the idea of the ‘survival of the fittest by generating solutions to problems. They are one of the tools we can use to apply machine learning to finding good, sometimes even optimal, solutions to problems that have huge number of potential solutions. They use biological processes in software to find answers to problems that have really large search spaces by continuously generating candidate solutions, evaluating how well the solutions fit the desired outcome, and refining the best solutions.

2. OUTLINE OF BASIC GENETIC ALGORITHM

Start: Generate random population of n chromosomes (suitable solutions for the problem)

Fitness: Evaluate the fitness f(x) of each chromosome x in the population

New population: Create a new population by repeating following steps until the new population is complete

Selection: Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to be selected)

Crossover: With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.

Mutation: With a mutation probability mutate new offspring at each locus (position in chromosome).

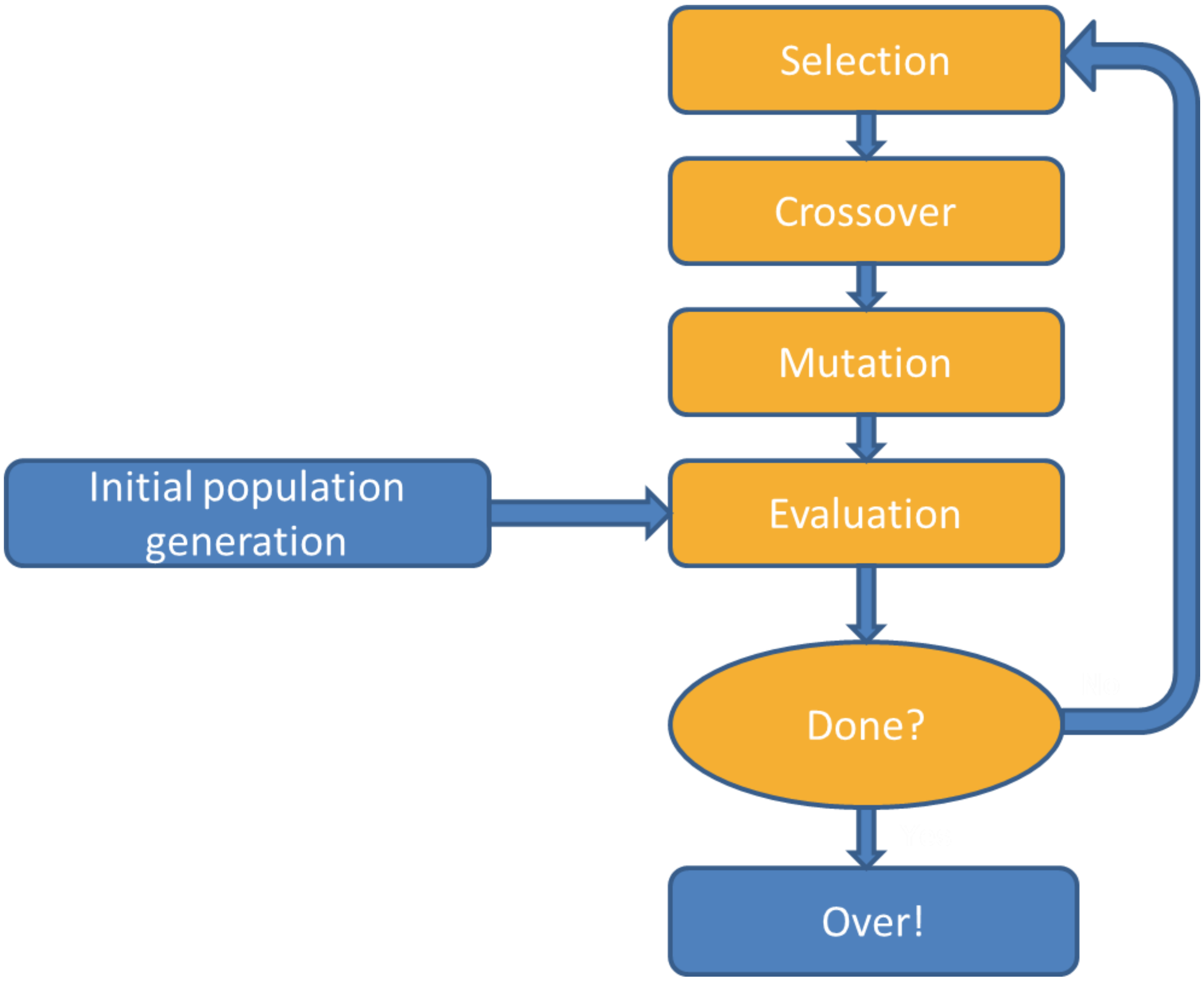
Accepting: Place new offspring in a new population

Replace: Use new generated population for a further run of algorithm

Test: If the end condition is satisfied, stop, and return the best solution in current population

Loop Go to Fitness step

We can use the schema below to simplify the GA flow



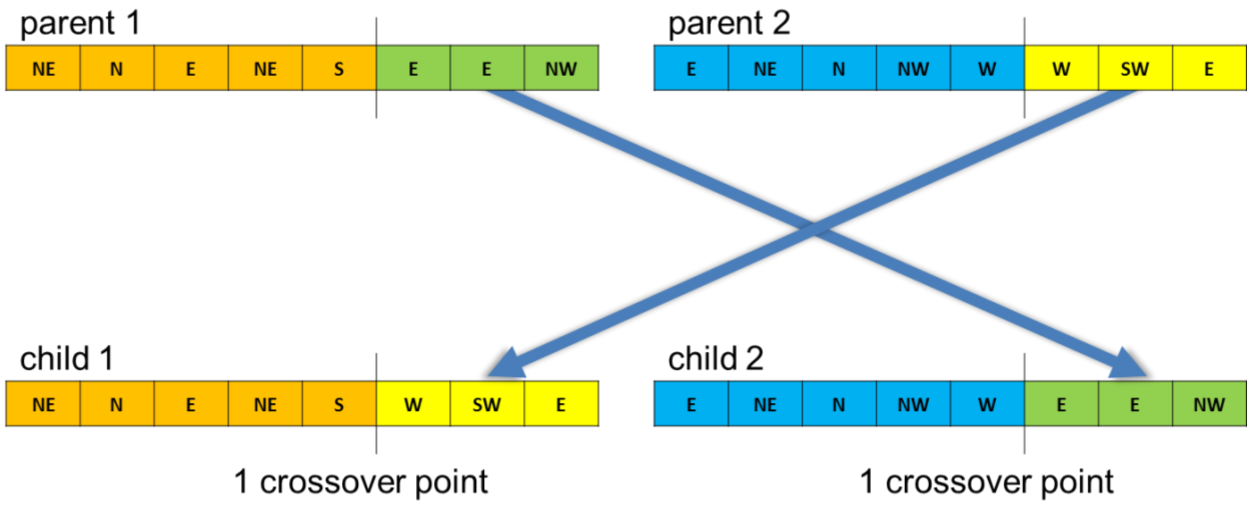
These algorithms reflect the natural selection process where populations of individuals create child populations, through genes crossover and mutation. The principle of a genetic algorithm is to evaluate these individuals, at each generation, in order to retrieve an elite which will represent the solution to the initial problem.

Detailed description:

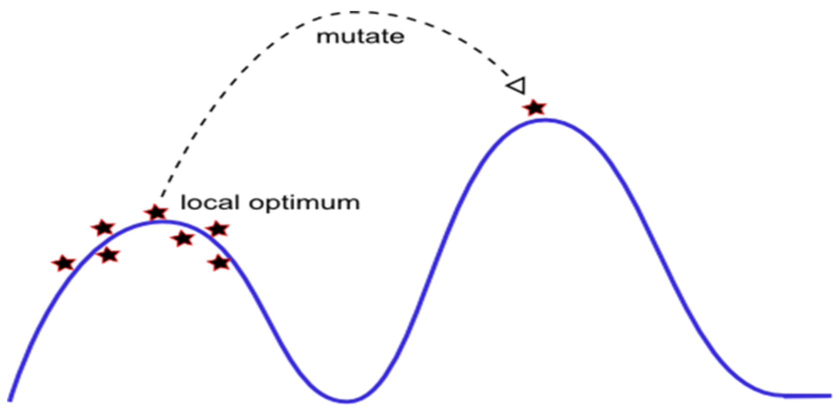
Crossover

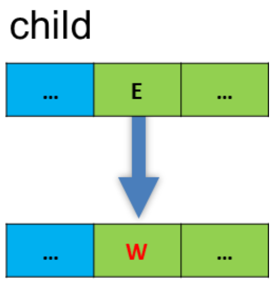
During crossover, all genes beyond one (or more) point(s) are swapped between the two parent organisms. There are many ways to do crossover

Here is an example with one crossover point:



Mutation
Gene mutation is when its value is replaced by another, chosen randomly. We refer here to a mutation rate. As an example, a 5% mutation rate means that a gene has 5% probability to be mutated, at each generation.





Graph extracted from http://www.codeproject.com/Articles/707505/Genetic-Algorithms-Demystified

Evaluation
Evaluation allows genetic algorithms to determine the best individuals at each generation. These individuals can therefore be selected in different ways, according to the current selection method, as parents for the next generation.
Selection
Different types of selection actually exist (roulette wheel, rank, tournament, etc.). Here, we will use the tournament selection in this python code. Basically, every two individuals are compared to each other. The strongest individual then has a probability of X % to be selected. The selection rate should be rather high (more than 70%) in order to favor best individuals, while giving a chance to the worst ones to be selected for the next generation.
The configuration of all these steps (crossover, mutation, selection, evaluation) is quite important

since it will impact the simulation global result. As an example, if the number of crossover points, or the mutation rate, is set too high, the population might be too diverse, and the algorithm might diverge. On the other hand, a too low mutation rate might lead the population into local optimum
Finally, the selection method must be adapted to the actual individual's fitness scores. As an instance, roulette wheel selection does not fit well with high standard deviation population scores, since fittest individuals will be continuously selected across generations.

3. LITERATURE REVIEW AND BACKGROUND

The traditional theory of GAs (first formulated in Holland 1975) assumes that, at a very general level of description, GAs work by discovering, emphasizing, and recombining good "building blocks" of solutions in a highly parallel fashion. The idea here is that good solutions tend to be made up of good building blocks—combinations of bit values that confer higher fitness on the strings in which they are present. Holland (1975) introduced the notion of *schemas* (or *schemata*) to formalize the informal notion of "building blocks." A schema is a set of bit strings that can be described by a template made up of ones, zeros, and asterisks, the asterisks representing wild cards (or "don't cares"). For example, the schema $H = 1 * * * * 1$ represents the set of all 6−bit strings that begin and end with 1. Goldberg (1989a) refers to H as "hyperplane." H is used to denote schemas because schemas define hyperplanes, "planes" of various dimensions in the l dimensional space of length−l bit strings.) The strings that fit this template (e.g., 100111 and 110011) are said to be *instances* of *H*. The schema H is said to have two *defined* bits (non-asterisks) or, equivalently, to be of *order* 2. Its *defining length* (the distance between its outermost defined bits) is 5. Here the term "schema" is used to denote both a subset of strings represented by such a template and the template itself. A central tenet of traditional GA theory is that schemas are implicitly the building blocks that the GA processes effectively under the operators of selection, mutation, and single point crossover. In evaluating a population of n strings, the GA is implicitly estimating the average fitness's of all schemas that are present in the population, and increasing or decreasing their representation according to the Schema Theorem. This simultaneous implicit evaluation of large numbers of schemas in a population of n strings is known as *implicit parallelism* (Holland 1975). The effect of selection is to gradually bias the sampling procedure toward instances of schemas whose fitness is estimated to be above average. Over time, the estimate of a schema's average fitness should, in principle, become more and more accurate since the GA is sampling more and more instances of that schema.

The Schema Theorem and the Building Block Hypothesis deal primarily with the roles of selection and crossover in GAs. What is the role of mutation? Holland (1975) proposed that mutation is what prevents the loss of diversity at a given bit position. For example, without mutation, every string in the population might come to have a one at the first bit position, and there would then be no way to obtain a string beginning with a zero. Mutation provides an "insurance policy" against such fixation. GA researchers have defined other types of crossover operators that deal with different types of building blocks, and have analyzed the generalized "schemas" that a given crossover operator effectively manipulates (Radcliffe 1991; Vose 1991). Once the population begins to converge at some local, the samples of some schemas are no longer uniform. For example, suppose instances of 111 *⋯* are highly fit and the population has more or less converged on those bits (i.e., nearly every member of the population is an instance of that schema). Then almost all samples of, say ***000 *⋯* will actually be samples of 111000 *⋯*. This may prevent the GA from making any accurate estimate of *u* (* * *000 *⋯*).

If a schema's static average fitness has high variance, the GA may not be able to make an accurate estimate of this static average fitness.

3.1. SELECTION METHODS

After deciding on an encoding, the second decision to make in using a genetic algorithm is how to perform selection—that is, how to choose the individuals in the population that will create offspring for the next generation, and how many offspring each will create. The purpose of selection is, of course, to emphasize the fitter individuals in the population in hopes that their offspring will in turn have even higher fitness. Selection has to be balanced with variation from crossover and mutation (the "exploitation/exploration balance"): too strong selection means that suboptimal highly fit individuals will take over the population, reducing the diversity needed for further change and progress; too weak selection will result in too slow evolution. Many selection schemes have been proposed in the GA literature.

3.1.1. Fitness−Proportionate Selection with "Roulette Wheel" and "Stochastic Universal" Sampling

Holland's original GA used fitness proportionate selection, in which the "expected value" of an individual (i.e., the expected number of times an individual will be selected to reproduce) is that individual's fitness divided by the average fitness of the population. The most common method for implementing this is "roulette wheel" sampling: each individual is assigned a slice of a circular "roulette wheel," the size of the slice being proportional to the individual's fitness. The wheel is spun *N* times, where *N* is the number of individuals in the population. On each spin, the individual under the wheel's marker is selected to be in the pool of parents for the next generation.

James Baker (1987) proposed a different sampling method, "stochastic universal sampling" (SUS) to minimize this "spread" (the range of possible actual values, given an expected value). Rather than spin the roulette wheel *N* times to select *N* parents, SUS spins the wheel once, but with *N* equally spaced pointers, which are used to selected the *N* parents.

3.1.2. Sigma Scaling

To solve some of the issues with the methods above such as premature convergence, GA researchers have experimented with several "scaling" methods for mapping "raw" fitness values to expected values so as to make the GA less susceptible to premature convergence. One example is "sigma scaling" (Forrest 1985; it was called "sigma truncation" in Goldberg 1989a), which keeps the selection pressure (i.e., the degree to which highly fit individuals are allowed many offspring) relatively constant over the course of the run rather than depending on the fitness variances in the population. Under sigma scaling, an individual's expected value is a function of its fitness, the population mean, and the population standard deviation.

3.1.3. Elitism

Elitism, first introduced by Kenneth De Jong (1975), is an addition to many selection methods that forces the GA to retain some number of the best individuals at each generation. Such individuals can be lost if they are not selected to reproduce or if they are destroyed by crossover or mutation. Many researchers have found that elitism significantly improves the GA's performance.

3.1.4. Boltzmann Selection

Sigma scaling keeps the selection pressure more constant over a run. But often different amounts of selection pressure are needed at different times in a run for example, early on it might be good to be liberal, allowing less fit individuals to reproduce at close to the rate of fitter individuals, and having selection occur slowly while maintaining a lot of variation in the population. Later it might be good to have selection be stronger in order to strongly emphasize highly fit individuals, assuming that the early diversity with slow selection has allowed the population to find the right part of the search space. Fitness proportionate selection is commonly used in GAs mainly because it was part of Holland's original proposal and because it is used in the Schema Theorem, but, evidently, for many applications simple fitness proportionate selection requires several "fixes" to make it work well. In recent years completely different approaches to selection (e.g., rank and tournament selection) have become increasingly common.

3.1.4. Rank Selection

Rank selection is an alternative method whose purpose is also to prevent too quick convergence. In the version proposed by Baker (1985), the individuals in the population are ranked according to fitness, and the expected value of each individual depends on its rank rather than on its absolute fitness. There is no need to scale fitness in this case, since absolute differences in fitness are obscured. This discarding of absolute fitness information can have advantages (using absolute fitness can lead to convergence problems) and disadvantages (in some cases it might be important to know that one individual is far fitter than its nearest competitor). Ranking avoids giving the far largest share of offspring to a small group of highly fit individuals, and thus reduces the selection pressure when the fitness variance is high.

3.1.5. Tournament Selection

The fitness proportionate methods described above require two passes through the population at each generation: one pass to compute the mean fitness (and, for sigma scaling, the standard deviation) and one pass to compute the expected value of each individual. Rank scaling requires sorting the entire population by rank a potentially time-consuming procedure. Tournament selection is similar to rank selection in terms of selection pressure, but it is computationally more efficient and more amenable to parallel implementation.

Two individuals are chosen at random from the population. A random number r is then chosen between 0 and 1. If $r < k$ (where k is a parameter, for example 0.75), the fitter of the two individuals is selected to be a parent; otherwise the less fit individual is selected. The two are then returned to the original population and can be selected again. An analysis of this method was presented by Goldberg and Deb (1991).

3.1.6. Steady State Selection

Most GAs described in the literature have been "generational" at each generation the new population consists entirely of offspring formed by parents in the previous generation (though some of these offspring may be identical to their parents). In some schemes, such as the elitist schemes described above, successive generations overlap to some degree, some portion of the previous generation is retained in the new population. The fraction of new individuals at each generation has been called the "generation gap" (De Jong 1975). In steady state selection, only a few individuals are replaced in each generation: usually a small number of the least fit individuals are replaced by offspring resulting from crossover and mutation of the fittest individuals. Steady-state GAs are often used in evolving rule based systems (e.g., classifier systems; see Holland 1986) in which incremental learning (and remembering what has already been learned) is important and in which members of the population collectively (rather than individually) solve the problem at hand. Steady-state selection has been analyzed by Syswerda (1989, 1991), by Whitley (1989), and by De Jong and Sarma (1993).

3.2. GENETIC OPERATORS

3.2.1. Crossover

It could be said that the main distinguishing feature of a GA is the use of crossover. Single-point crossover is the simplest form: a single crossover position is chosen at random and the parts of two parents after the crossover position are exchanged to form two offspring. The idea here is, of course, to recombine building blocks (schemas) on different strings. Single point crossover has some shortcomings, though. For one thing, it cannot combine all possible schemas. For example, it cannot in general combine instances of 11*****1 and ****11** to form an instance of 11**11*1. Likewise, schemas with long defining lengths are likely to be destroyed under single point crossover. Eshelman, Caruana, and Schaffer (1989) call this "positional bias": the schemas that can be created or destroyed by a crossover depend strongly on the location of the bits in the chromosome. Single point crossover assumes that short, low order schemas are the functional building blocks of strings, but one generally does not know in advance what ordering of bits will group functionally related bits together, this was the purpose of the inversion operator and other adaptive operators described above. Eshelman, Caruana, and Schaffer also point out that there may not be any way to put all functionally related bits close together on a string, since particular bits might be crucial in more than one schema. They point out further that the tendency of single point crossover to keep short schemas intact can lead to the preservation of hitchhikers' bits that are not part of a desired schema but which, by being close on the string, hitchhike along with the beneficial schema as it reproduces

To reduce positional bias and this "endpoint" effect, many GA practitioners use two-point crossover, in which two positions are chosen at random and the segments between them are exchanged. Two-point crossover is less likely to disrupt schemas with large defining lengths and can combine more schemas than single-point crossover. In addition, the segments that are exchanged do not necessarily contain the endpoints of the strings. Again, there are schemas that two–point crossover cannot combine. GA practitioners have experimented with different numbers of crossover points (in one method, the number of crossover points for each pair of parents is chosen from a Poisson distribution whose mean is a function of the length of the chromosome). Some practitioners (e.g., Spears and De Jong (1991)) believe strongly in the superiority of "parameterized uniform crossover," in which an exchange happens at each bit position with probability *p* (typically $0.5 < p < 0.8$). Parameterized uniform crossover has no positional bias, any schemas contained at different positions in the parents can potentially be recombined in the offspring.

3.2.2 Mutation

A common view in the GA community, dating back to Holland's book *Adaptation in Natural and Artificial Systems*, is that crossover is the major instrument of variation and innovation in GAs, with mutation insuring the population against permanent fixation at any particular locus and thus playing more of a background role. This differs from the traditional positions of other evolutionary computation methods, such as evolutionary programming and early versions of evolution strategies, in which random mutation is the only source of variation. (Later versions of evolution strategies have included a form of crossover.)

However, the appreciation of the role of mutation is growing as the GA community attempts to understand how GAs solve complex problems. Some comparative studies have been performed on the power of mutation versus crossover; for example, Spears (1993) formally verified the intuitive idea that, while mutation and crossover have the same ability for "disruption" of existing schemas, crossover is a more robust "constructor" of new schemas. Mühlenbein (1992, p. 15), on the other hand, argues that in many cases a hill-climbing strategy will work better than a GA with crossover and that "the power of mutation has been underestimated in traditional genetic algorithms

3.2.3. Other Operators and Mating Strategies

Though most GA applications use only crossover and mutation, many other operators and strategies for applying them have been explored in the GA literature. These include inversion and gene doubling (discussed above) and several operators for preserving diversity in the population. For example, De Jong (1975) experimented with a "crowding" operator in which a newly formed offspring replaced the existing individual most similar to itself. This prevented too many similar individuals ("crowds") from being in the population at the same time. Goldberg and Richardson (1987) accomplished a similar result using an explicit "fitness sharing" function: each individual's fitness was decreased by the presence of other population members, where the amount of decrease due to each other population member was an explicit increasing function of the similarity between the two individuals. Thus, individuals that were similar to many other individuals were punished, and individuals that were different were rewarded. Goldberg and Richardson showed that in some cases this could induce appropriate "speciation," allowing the population members to converge on several peaks in the fitness landscape rather than all converging to the same peak. Smith, Forrest, and Perelson (1993) showed that a similar effect could be obtained without the presence of an explicit sharing function.

A different way to promote diversity is to put restrictions on mating. For example, if only sufficiently similar individuals are allowed to mate, distinct "species" (mating groups) will tend to form. This approach has been studied by Deb and Goldberg (1989). Eshelman (1991) and Eshelman and Schaffer (1991) used the opposite tack: they disallowed matings between sufficiently similar individuals ("incest"). Their desire was not to form species but rather to keep the entire population as diverse as possible. Holland (1975) and Booker (1985) have suggested using "mating tags" parts of the chromosome that identify prospective mates to one another. Only those individuals with matching tags are allowed to mate (a kind of "sexual selection" procedure). These tags would, in principle, evolve along with the rest of the chromosome to adaptively implement appropriate restrictions on mating. Finally, there have been some experiments with spatially restricted mating (see, e.g., Hillis 1992): the population evolves on a spatial lattice, and individuals are likely to mate only with individuals in their spatial neighborhoods. Hillis found that such a scheme helped preserve diversity by maintaining spatially isolated species, with innovations largely occurring at the boundaries between species.

4. PROBLEM STATEMENT

The program will prompt a user to enter a text, the program will use Genetic Algorithm to return the initial text entered.

Depending of the parameters of the programs: mutation rate, crossover rate, size of text, the program can succeed in predicting the initial text and it can fail at predicting it.

The measures of the fitness through different generations are printed in the outputs: Best of Fitness, Average of Fitness and Worst of Fitness.

The initial text entered in the computer is only used for guidance to the algorithm as optimal value to be considered as fitness value.

5. METHODOLOGY

Program flow:

Randomly initialize a population of chromosomes
Any combination of characters that meet the length of the optimal text is a string of chromosomes
Compute fitness: done by the” run ()” function
next ()
Generate parents, this is done by the “next ()” function
Set the probability of crossover at a value you wish (0.9 for example)
While the size of the entire population is not reached:
Generate a random number if that number is less than the set probability crossover is performed on the string
Generate a random number, for each offspring mutate if the random number generated is less than the mutation rate
Parents function sorts each generation of chromosomes according to their fitness.
Tournament selection is used to choose parents

Crossover
Two-point crossover, done by the function crossover (). randomly pick two indexes as indexes, depending of the values of the two indexes break and attached two parents' strings at two points to generate two children.
Mutation
A random index less than the length of the target (optimal text) is chosen
Another random number between -5 and 5 is chosen as index
All the chromosomes are listed
The random indexes positions are used and the 2 characters are swap.
Selection type: Tournament selection is used to pick the best parents, this accomplish by using the Compete function
The tournament selection is done by using Compete() function
The function picks two elements at random from the population compute their fitness and return the element that get the higher fitness.
Encoding and decoding: ord() and ch() functions used inside the functions GenoToPheno() and PhenoTogGeno() handle the coding and decoding part. GenoTopheno function returns the numeric(coding) values associated to the text string by internally using ord() function
GenoToPheno ('abc')=979899PhenoToGeno function does exactly the opposite (decoding)
PhenoToGeno (989799)=bac

Example: ord('a') returns the integer 97 and ch(97) returns a, so text aaa is 979797

Fitness

The fitness values the genetic algorithm provides is the **only** feedback the program gets to guide it toward a solution. In this program, the fitness value is the total number of letters in the guess that match the letter in the same position of the password. It is done using function Fitness() which return the sum of c-t with c in chromosome and t in the initial text(optimal text) if both positions match and are the same letters .

6. RESULTS AND CONCLUSION

This interactive program helps greatly determine the importance of the parameters and operators in Genetic Algorithm.

As the crossover initial threshold probability decreases, the GA struggles to find the optimal solution as it reaches zero it is virtually impossible to generate to retrieve the initial text entered.

We can conclude that crossover is one of main qualities genetic algorithms need, so we come to an agreement with the previous authors that emphasis on the need of crossover for the GA to converge.

Mutation rate in the other hand improves the predictions as the value decreases, when the mutation rate increases, the algorithm find it hard to reach to the solution. At mutation rate of 1, the GA fails to predict the initial text. This result matches the intuition in the sense that if all characters are randomly switch, getting the solution is obviously impossible.

The complexity of the text also plays a role on the algorithm performance, for longer text, it takes more generations for the algorithm to reach while for shorter text, it is faster. So, performance of the algorithm is related to the complexity of the problem.

The number of generations allowed to be reached and the size of the population are also important. The population size should be large enough to contain all combinations possible of the initial text entered. The number of maximum generation as large as it gets, give more space to the GA to perform and converge to the right solution.

7. FUTURE WORK AND RECOMMENDATIONS

For future externalizing the fitness function will be explored and evaluation of the algorithm performance will be monitored to see if there is any improvement or relationship.

Prediction of pictures based on GA will be explored also as a possibility!

Developing a hybrid system based on GA and neural network will be addressed for future work.

In term of recommendation, depending on the problem at hand and depending on how easy the encoding and decoding part of the problems, GA could be used to efficiently solve problems but one must tune the parameters and operators' values to get good results

8. REFERENCES

Bäck, T. 1996. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms.* Oxford.

Belew, R. K. and Booker, L. B., eds. 1991. *Proceedings of the Fourth International Conference on Genetic Algorithms.* Morgan Kaufmann.

Davis, L., ed. 1987. *Genetic Algorithms and Simulated Annealing.* Morgan Kaufmann. Davis, L., ed. 1987. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold.

Eshelman, L. J., ed. 1995. *Proceedings of the Sixth International Conference on Genetic Algorithms* Morgan Kaufmann.

Fogel, D. B. 1995. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence.* IEEE Press.

Forrest, S., ed. 1993. *Proceedings of the Fifth International Conference on Genetic Algorithms.* Morgan Kaufmann.

Grefenstette, J. J., ed. 1985. *Proceedings of an International Conference on Genetic Algorithms and Their Applications.* Erlbaum.

Grefenstette, J. J., ed. 1987. *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms.* Erlbaum.

Goldberg, D. E. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison−Wesley. Holland, J. H. 1975. *Adaptation in Natural and Artificial Systems.* University of Michigan Press. (Second edition: MIT Press, 1992.)

Michalewicz, Z. 1992. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer−Verlag.

Rawlins, G., ed. 1991. *Foundations of Genetic Algorithms.* Morgan Kaufmann.

Schaffer, J. D., ed. 1989. *Proceedings of the Third International Conference on Genetic Algorithms.* Morgan Kaufmann.

Schwefel, H.−P. 1995. *Evolution and Optimum Seeking.* Wiley. Whitley, D., ed. 1993. *Foundations of Genetic Algorithmsc 2.* Morgan Kaufmann. Whitley, D., and Vose, M., eds. 1995. *Foundations of Genetic Algorithms 3.* Morgan Kaufmann.

Clinton Sheppard, *Introduction to Genetic Algorithms with Python.*

M. Melanie, An Introduction to Genetic Algorithms, Massachusetts: MIT Press, 1999.

9. APPENDIX

9.1 Python Code

```python
import random
class GA(object):
    def __init__(self, Genes):
        self.Genes = Genes
        pass

    def run(self):
        POP = self.Genes.initial()
        while True:
            fitPOP = [(self.Genes.Fitness(ch),  ch) for ch in POP]
            if self.Genes.Check(fitPOP): break
            POP = self.next(fitPOP)
            pass
        return POP

    def next(self, fits):
        generateParent = self.Genes.Parent(fits)
        size = len(fits)
        nexts = []
        while len(nexts) < size:
            Parent = next(generateParent)
            cross = random.random() < self.Genes.crossoverProba()
            children = self.Genes.crossover(Parent) if cross else Parent
            for ch in children:
                mutate = random.random() < self.Genes.mutationProba()
                nexts.append(self.Genes.mutation(ch) if mutate else ch)
                pass
            pass
        return nexts[0:size]
    pass

class functionGenes(object):
    def crossoverProba(self):#return crossover rate
        return 1.0

    def mutationProba(self):
        return 0.0

    def initial(self):
        return []

    def Fitness(self, chromosome):
        return len(chromosome)

    def Check(self, fitPOP):#stop run if returns true,list of fit populations
        return False

    def Parent(self, fitPOP):
        gen = iter(sorted(fitPOP))
        while True:
            f1, ch1 = next(gen)
            f2, ch2 = next(gen)
            yield (ch1, ch2)
            pass
        return
    def crossover(self, Parent):
        return Parent

    def mutation(self, chromosome):
        return chromosome
    pass

if __name__ == "__main__":

    class predictText(functionGenes):
        def __init__(self, Text,
                          limit=eval(input("Enter the Maximum number of
Generations\n ")), size=eval(input("The maximum length of characters?\n")),
                          prob_crossover=eval(input("Enter your probability of
crossover\n")), prob_mutation=eval(input("Enter your probability of
mutation\n"))):
            self.optimalText = self.GenoToPheno(Text)
            self.counter = 0

            self.limit = limit
            self.size = size
            self.prob_crossover = prob_crossover
            self.prob_mutation = prob_mutation
            pass

        def crossoverProba(self):
            return self.prob_crossover

        def mutationProba(self):
            return self.prob_mutation

        def initial(self):
            return [self.randomChromosome() for j in range(self.size)]

        def Fitness(self, chromo):
            return -sum(abs(c - t) for c, t in zip(chromo, self.optimalText))

        def Check(self, fitPOP):
            self.counter += 1
            if self.counter % 10 == 0:
                best_match = list(sorted(fitPOP))[-1][1]
                fits = [f for f, ch in fitPOP]
                best = max(fits)
                worst = min(fits)
                ave = sum(fits) / len(fits)
                print(
                    "[G %3d] Fitness=(Best=%4d,Avg= %4d, Worst=%4d): %r" %
                    (self.counter, best, ave, worst,
                     self.PhenoToGeno(best_match)))
                pass
            return self.counter >= self.limit

        def Parent(self, fitPOP):
            while True:
                Parent1 = self.Compete(fitPOP)
                Parent2 = self.Compete(fitPOP)
                yield (Parent1, Parent2)
                pass
            pass

        def crossover(self, Parent):
            Parent1, Parent2 = Parent
            index1 = random.randint(1, len(self.optimalText) - 2)
            index2 = random.randint(1, len(self.optimalText) - 2)
            if index1 > index2: index1, index2 = index2, index1
            child1 = Parent1[:index1] + Parent2[index1:index2] +
Parent1[index2:]
            child2 = Parent2[:index1] + Parent1[index1:index2] +
Parent2[index2:]
            return (child1, child2)

        def mutation(self, chromosome):
            index = random.randint(0, len(self.optimalText) - 1)
            vary = random.randint(-5, 5)
            mutated = list(chromosome)
            mutated[index] += vary
            return mutated

        def Compete(self, fitPOP):
            Firstf, First = self.randomSelection(fitPOP)
            Secondf, Second = self.randomSelection(fitPOP)
            return First if Firstf > Secondf else Second

        def randomSelection(self, fitPOP):
            return fitPOP[random.randint(0, len(fitPOP)-1)]

        def GenoToPheno(self, text):
            return [ord(ch) for ch in text]
        def PhenoToGeno(self, chromo):
            return "".join(chr(max(1, min(ch, 255))) for ch in chromo)
        def randomChromosome(self):
            return [random.randint(1, 255) for i in
range(len(self.optimalText))]
        pass

    GA(predictText(input("Enter your words here\n"))).run()
```

9.2 Outputs:

9.2.1 Here GA failed to return the correct text' Binghamton is a great school' after 100 generations

```
runfile('/Users/dieudonneouedraogo/Downloads/SSIE 519/Dieudonne_GeneticClean1.py',
wdir='/Users/dieudonneouedraogo/Downloads/SSIE 519')

Enter the Maximum number of Generations

 100

The maximum length of characters?

400

Enter your probability of crossover

0.9

Enter your probability of mutation

0.05

Enter your words here

Binghamton is a great school

[G  10] Fitness=(Best=-597,Avg= -1095, Worst=-1484): '\x04xu\x84[a_q\x80j\x07gi=¥H+col]6rg>Z\x85]'

[G  20] Fitness=(Best=-338,Avg= -488, Worst=-665): 'HSurTZgq\x80d!fw)R\x1dqg\x7fY\x8d6udsfw©'

[G  30] Fitness=(Best=-242,Avg= -278, Worst=-336): 'HlurlZgr\x80d!fw)R\x1d_qce{6rcsfw©'
```

```
[G  40] Fitness=(Best=-194,Avg= -213, Worst=-233): 'HlpilZgqqj!hw\x1dU\x1d_qceq6rdsfw©'
[G  50] Fitness=(Best=-168,Avg= -182, Worst=-193): 'HlpilZoqqn!hw\x1dX\x1d_qceq-rdshs©'
[G  60] Fitness=(Best=-149,Avg= -160, Worst=-170): 'HlpilZoqqn!ht\x1d\\#_qcaq*rdnhs©'
[G  70] Fitness=(Best=-131,Avg= -140, Worst=-151): 'HjnijZlvqn!ht\x1d\\#_qcaq%ucnhq¦'
[G  80] Fitness=(Best=-111,Avg= -124, Worst=-134): 'Cjnih^oqqn!ht\x1da#_qcaq rcnhq¦'
[G  90] Fitness=(Best= -96,Avg= -104, Worst=-114): 'Hjnfhblvqn ht\x1f`"cqcaq rcnhn\x9f'
[G 100] Fitness=(Best= -79,Avg=  -88, Worst=-100): 'Hjnih_lvqn ht!a"fqeaq rdjmn\x9a'
```

9.2.2 .Here it's failed after 100 generation to predict '**Dieudonne.O.N**'

```
runfile('/Users/dieudonneouedraogo/Downloads/SSIE 519/Dieudonne_GeneticClean1.py',
wdir='/Users/dieudonneouedraogo/Downloads/SSIE 519')

Enter the Maximum number of Generations
 100

The maximum length of characters?
400

Enter your probability of crossover
0.9

Enter your probability of mutation
0.05

Enter your words here
Dieudonne.O.N
[G  10] Fitness=(Best=-219,Avg= -395, Worst=-586): 'KFcmaYv¤e>`VU'
[G  20] Fitness=(Best=-107,Avg= -164, Worst=-216): 'Kh\\uasvoe8`VU'
[G  30] Fitness=(Best= -74,Avg=  -96, Worst=-117): 'Cievahvoe0G)t'
```

```
[G  40] Fitness=(Best= -58,Avg=  -65, Worst= -77): 'Cievasloe0T-t'

[G  50] Fitness=(Best= -46,Avg=  -52, Worst= -60): 'Cieuesnoe0P.r'

[G  60] Fitness=(Best= -37,Avg=  -41, Worst= -50): 'Cieueonoe0O-m'

[G  70] Fitness=(Best= -31,Avg=  -34, Worst= -40): 'Cieudonoe.O-j'

[G  80] Fitness=(Best= -27,Avg=  -29, Worst= -37): 'Cieueonoe.O-e'

[G  90] Fitness=(Best= -23,Avg=  -25, Worst= -32): 'Cieudonne.O.d'

[G 100] Fitness=(Best= -18,Avg=  -22, Worst= -28): 'Cieudonne.O._'
```

9.2.3 here it's succeed at predicting 'Dieudonne Ouedraogo' after 120 generations

```
runfile('/Users/dieudonneouedraogo/Downloads/SSIE 519/Dieudonne_GeneticClean1.py',
wdir='/Users/dieudonneouedraogo/Downloads/SSIE 519')


Enter the Maximum number of Generations

  200


The maximum length of characters?

400


Enter your probability of crossover

0.9


Enter your probability of mutation

0.2


Enter your words here

Dieudonne Ouedraogo

[G  10] Fitness=(Best=-283,Avg= -595, Worst=-957): '\\sk\x89b\x84qwh/QxgVoh\x06v\\'

[G  20] Fitness=(Best=-141,Avg= -210, Worst=-289): '\\sktirqpQ!_yh]s]qva'

[G  30] Fitness=(Best= -84,Avg= -115, Worst=-140): 'difsbrqoh!Qxh]s]tdw'
```

```
[G  40] Fitness=(Best= -64,Avg=  -76, Worst= -90): '`iftboqoh!Mxhhr]qgu'

[G  50] Fitness=(Best= -45,Avg=  -55, Worst= -70): '[iftbmqpe!Otcdrangu'

[G  60] Fitness=(Best= -30,Avg=  -40, Worst= -50): 'Xifteonoe!Oucdrangq'

[G  70] Fitness=(Best= -27,Avg=  -32, Worst= -40): 'Xieteonoe!Ouddraogq'

[G  80] Fitness=(Best= -16,Avg=  -22, Worst= -33): 'Nidtconne!Ouedraogq'

[G  90] Fitness=(Best=  -9,Avg=  -15, Worst= -23): 'Iiftdonne Ouedraogq'

[G 100] Fitness=(Best=  -4,Avg=   -9, Worst= -16): 'Eietdonne Ouedraogq'

[G 110] Fitness=(Best=  -1,Avg=   -4, Worst= -10): 'Eieudonne Ouedraogo'

[G 120] Fitness=(Best=   0,Avg=   -1, Worst=  -9): 'Dieudonne Ouedraogo'

[G 130] Fitness=(Best=   0,Avg=    0, Worst=  -6): 'Dieudonne Ouedraogo'

[G 140] Fitness=(Best=   0,Avg=    0, Worst=  -7): 'Dieudonne Ouedraogo'

[G 150] Fitness=(Best=   0,Avg=    0, Worst=  -6): 'Dieudonne Ouedraogo'

[G 160] Fitness=(Best=   0,Avg=    0, Worst=  -6): 'Dieudonne Ouedraogo'

[G 170] Fitness=(Best=   0,Avg=    0, Worst=  -5): 'Dieudonne Ouedraogo'

[G 180] Fitness=(Best=   0,Avg=    0, Worst=  -5): 'Dieudonne Ouedraogo'

[G 190] Fitness=(Best=   0,Avg=    0, Worst=  -8): 'Dieudonne Ouedraogo'

[G 200] Fitness=(Best=   0,Avg=    0, Worst=  -6): 'Dieudonne Ouedraogo'
```

9.2.4 without crossover the GA is not able to predict the text

```
runfile('/Users/dieudonneouedraogo/Downloads/SSIE 519/Dieudonne_GeneticClean1.py',
wdir='/Users/dieudonneouedraogo/Downloads/SSIE 519')


Enter the Maximum number of Generations

 400


The maximum length of characters?

400
```

```
Enter your probability of crossover

0.0


Enter your probability of mutation

0.2


Enter your words here

dieudonne ouedraogo

[G  10] Fitness=(Best=-794,Avg= -811, Worst=-911): '9]V\x894}Na¯¯\x8fsT@o\x84áÜW'

[G  20] Fitness=(Best=-777,Avg= -787, Worst=-795): 'Ud¥vS7Sjd<7\x8aù7\x83\x91²âM'

[G  30] Fitness=(Best=-758,Avg= -771, Worst=-781): 'Z`\xa0vS7Shd<B\x89ù<\x88\x91³ÞM'

[G  40] Fitness=(Best=-738,Avg= -750, Worst=-759): ']`\xa0vS@Shd<=\x89õ<\x88\x8d³ÞR'

[G  50] Fitness=(Best=-722,Avg= -733, Worst=-741): 'Z`\x9dvX7Sjd7F\x89õ<\x81\x90®ÞM'

[G  60] Fitness=(Best=-703,Avg= -715, Worst=-726): 'Z`\x9dvX7Xjd7F\x84õ<|\x90®ÚM'

[G  70] Fitness=(Best=-691,Avg= -701, Worst=-709): 'cg\xa0vS7Xld<D\x86õF\x81\x8b®ÝQ'

[G  80] Fitness=(Best=-675,Avg= -684, Worst=-691): 'cg\xa0vS7Xld<I\x86õH}\x8b®ÝV'

[G  90] Fitness=(Best=-661,Avg= -669, Worst=-679): 'cl\x9bvS9\\od7D\x81ñF\x81\x8b®ÝV'

[G 100] Fitness=(Best=-649,Avg= -656, Worst=-663): 'cg\x9bvS>\\ld<D\x82ñF}\x83©ÜQ'

[G 110] Fitness=(Best=-637,Avg= -643, Worst=-651): 'cg\x9bvS7Xnd<L\x81õO\x81\x83®Ø]'

[G 120] Fitness=(Best=-622,Avg= -630, Worst=-639): 'cg\x97vX<`nd;H\x81õF\x81\x83ªÐV'

[G 130] Fitness=(Best=-603,Avg= -614, Worst=-625): 'cg\x97vX@`nd;H\x81ðF}\x83¨ÐZ'

[G 140] Fitness=(Best=-592,Avg= -600, Worst=-609): 'cg\x97vX@end;H\x81ðFx\x83¨Ð['

[G 150] Fitness=(Best=-574,Avg= -584, Worst=-593): 'cg\x89v\\@`nd6H\x81ðFx\x83¨Ð['

[G 160] Fitness=(Best=-562,Avg= -570, Worst=-578): 'cg\x8avX@`nd1H\x81ðJx~¨Ð^'

[G 170] Fitness=(Best=-549,Avg= -556, Worst=-564): 'cg\x84sa@cnd6L\x81ìKx\x83¨Ð['

[G 180] Fitness=(Best=-533,Avg= -542, Worst=-551): 'cg\x8evXHjnd5L|ðFs~\xa0Ï^'

[G 190] Fitness=(Best=-519,Avg= -528, Worst=-537): 'ci\x84sbIhnd3M\x81ìFs\x7f¨Ë['

[G 200] Fitness=(Best=-507,Avg= -515, Worst=-523): 'ci\x80sbIhnd3U\x81ìFs\x7f¨Ë['

[G 210] Fitness=(Best=-493,Avg= -502, Worst=-511): 'ci\x7fsbLfnd4O\x81êJx\x81¢Æd'
```

```
[G 220] Fitness=(Best=-481,Avg= -489, Worst=-498): 'cj\x84vXKgnd.HwìQs{¤Çi'

[G 230] Fitness=(Best=-469,Avg= -476, Worst=-484): 'cg\x84vbKjnd.HwìQs{¤Çi'

[G 240] Fitness=(Best=-451,Avg= -462, Worst=-473): 'ci\x7fwbVgnd6S\x81åRx\x7f\x9cÆl'

[G 250] Fitness=(Best=-438,Avg= -448, Worst=-457): 'ci\x7fwbVjnd6S|åRx\x7f\x9cÁl'

[G 260] Fitness=(Best=-426,Avg= -434, Worst=-444): 'ci\x7fwb_gnd2S\x81ÝNxw\x97Æg'

[G 270] Fitness=(Best=-413,Avg= -422, Worst=-429): 'ci\x7fwb^lnd0S~âJxw\x97Áo'

[G 280] Fitness=(Best=-399,Avg= -409, Worst=-418): 'cinwbZlnd2S\x81âJxw\x90Án'

[G 290] Fitness=(Best=-381,Avg= -392, Worst=-402): 'cinwbZlnd2S}ÝJxw\x87Án'

[G 300] Fitness=(Best=-371,Avg= -378, Worst=-386): 'cilwbZlnd2Z|âNxo\x8eÁn'

[G 310] Fitness=(Best=-360,Avg= -367, Worst=-375): 'dinub_lnd-ZyâJxs\x8d½n'

[G 320] Fitness=(Best=-345,Avg= -353, Worst=-362): 'cijwbZlne(SyÝKxk\x82Ám'

[G 330] Fitness=(Best=-334,Avg= -341, Worst=-349): 'diiubalnd(ZyâJso\x88½n'

[G 340] Fitness=(Best=-322,Avg= -329, Worst=-337): 'dihucelnd%ZyÞJxn\x89¹n'

[G 350] Fitness=(Best=-309,Avg= -318, Worst=-326): 'dihucelnd%_yÞJun\x84¹n'

[G 360] Fitness=(Best=-297,Avg= -304, Worst=-314): 'dihucelnd%^yÚJun\x80´n'

[G 370] Fitness=(Best=-285,Avg= -292, Worst=-300): 'diducilne%_yÚJun\x80°n'

[G 380] Fitness=(Best=-270,Avg= -281, Worst=-292): 'cimwf`lne,mvÞbsa\x88²p'

[G 390] Fitness=(Best=-259,Avg= -266, Worst=-275): 'cidwf`lne,mvÞbsa\x84²p'

[G 400] Fitness=(Best=-246,Avg= -257, Worst=-266): 'ciiwfilne,mvÙbsa\x83±p'
```

9.2.5 With mutation rate of 100% ,the GA struggles to predict the text

```
runfile('/Users/dieudonneouedraogo/Downloads/SSIE 519/Dieudonne_GeneticClean1.py', wdir='/Users/dieudonneouedraogo/Downloads/SSIE 519')


Enter the Maximum number of Generations

 400


The maximum length of characters?
```

```
400

Enter your probability of crossover
0.7

Enter your probability of mutation
1

Enter your words here
dieudonne ouedraogo
[G  10] Fitness=(Best=-423,Avg= -632, Worst=-902): '\\z=\x96xl}Zt@x\x81lo\x90cc\x94\x13'
[G  20] Fitness=(Best=-158,Avg= -293, Worst=-440): 'Zfc\x81kydtR\x18t\x84hjtkmxz'
[G  30] Fitness=(Best= -93,Avg= -129, Worst=-163): 'Zhdok\\nth!mvhjmcptu'
[G  40] Fitness=(Best= -57,Avg=  -77, Worst=-100): "\\hdxhupqe'muhfqbpmu"
[G  50] Fitness=(Best= -30,Avg=  -47, Worst= -66): 'gheugooqf muehqapku'
[G  60] Fitness=(Best= -15,Avg=  -31, Worst= -47): 'dkeucpond nuecraolq'
[G  70] Fitness=(Best= -10,Avg=  -21, Worst= -37): 'fketconoe osedraofo'
[G  80] Fitness=(Best=  -6,Avg=  -17, Worst= -27): 'cheucpmne oueeraogo'
[G  90] Fitness=(Best=  -6,Avg=  -15, Worst= -30): 'diesdonne ouedr`pio'
[G 100] Fitness=(Best=  -3,Avg=  -14, Worst= -29): 'dieuconne ouedr`ogn'
[G 110] Fitness=(Best=  -2,Avg=  -13, Worst= -25): 'dieudonne nuedrbogo'
[G 120] Fitness=(Best=  -5,Avg=  -13, Worst= -26): 'dieufonne\x1fouddraogp'
[G 130] Fitness=(Best=  -4,Avg=  -13, Worst= -28): 'dieudnnpe nuedraogo'
[G 140] Fitness=(Best=  -4,Avg=  -13, Worst= -25): 'eieudonnf owedraogo'
[G 150] Fitness=(Best=  -2,Avg=  -13, Worst= -29): 'cieudonne otedraogo'
[G 160] Fitness=(Best=  -4,Avg=  -13, Worst= -24): 'dhfudonnc ouedraogo'
[G 170] Fitness=(Best=  -2,Avg=  -13, Worst= -28): 'dievdonme ouedraogo'
[G 180] Fitness=(Best=  -4,Avg=  -13, Worst= -25): 'dieudonnc!ouedrango'
[G 190] Fitness=(Best=  -3,Avg=  -13, Worst= -28): 'dieudonmf ouedrapgo'
```

```
[G 200] Fitness=(Best=  -3,Avg=  -13, Worst= -25): 'dieudonne ouedramfo'
[G 210] Fitness=(Best=  -4,Avg=  -14, Worst= -25): 'dieseonne ouedrapgo'
[G 220] Fitness=(Best=  -2,Avg=  -13, Worst= -26): 'dieudonne ouedsapgo'
[G 230] Fitness=(Best=  -2,Avg=  -12, Worst= -27): 'dieudoone\x1fouedraogo'
[G 240] Fitness=(Best=  -2,Avg=  -12, Worst= -24): 'dieudnnme ouedraogo'
[G 250] Fitness=(Best=  -4,Avg=  -14, Worst= -26): 'eieudomoe ouedsaogo'
[G 260] Fitness=(Best=  -3,Avg=  -13, Worst= -26): 'eieudnnne ouedsaogo'
[G 270] Fitness=(Best=  -3,Avg=  -12, Worst= -24): 'dieudonoe\x1fnuedraogo'
[G 280] Fitness=(Best=  -4,Avg=  -13, Worst= -26): 'dieudonne ouedqangq'
[G 290] Fitness=(Best=  -3,Avg=  -13, Worst= -26): 'dieudonne\x1eouedsaogo'
[G 300] Fitness=(Best=  -4,Avg=  -13, Worst= -26): 'dieudonne nwecraogo'
[G 310] Fitness=(Best=  -4,Avg=  -12, Worst= -27): 'eidudonne ouefraogo'
[G 320] Fitness=(Best=  -4,Avg=  -13, Worst= -27): 'cieudopne ouedrango'
[G 330] Fitness=(Best=  -3,Avg=  -14, Worst= -24): 'dieudonne!ouecrapgo'
[G 340] Fitness=(Best=  -4,Avg=  -13, Worst= -29): 'eieudonne ouedraogl'
[G 350] Fitness=(Best=  -3,Avg=  -13, Worst= -28): 'dieuconne puedr`ogo'
[G 360] Fitness=(Best=  -4,Avg=  -13, Worst= -29): 'eieudpnne ouedrbofo'
[G 370] Fitness=(Best=  -2,Avg=  -13, Worst= -27): 'dieudonne ovedr`ogo'
[G 380] Fitness=(Best=   0,Avg=  -14, Worst= -26): 'dieudonne ouedraogo'
[G 390] Fitness=(Best=  -4,Avg=  -13, Worst= -25): 'cieudpnne!ouddraogo'
END !
```