ISSN No. 0976-5697

Volume 4, No. 9, July-August 2013

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Making Mutation Adaptive in Genetic Algorithm

Suyash Raghava
Department of Computer Science
Amity University Haryana Gurgaon, India
suyashraghava.research@gmail.com

Abstract: In classical Genetic Algorithm the nature of mutation is random so it only serves the purpose of adding diversity to the current generation and to avoid problems like premature convergence. In this paper it is shown that how mutation can be made adaptive so that when it occurs, it mutates the chromosome in a way so as to produce overall healthier chromosomes. The theory of adaptive mutation proposes that mutation may occur as a direct consequence of stress in the environment so that it can adapt to it. In this paper the mutation will follow up the theory of adaptive mutation and will try to mutate the chromosomes in a way so that it produces better results.

Keywords: Artificial Intelligence; Genetic Algorithm; Evolutionary Algorithm; Adaptive Mutation; Travelling Salesman Problem.

I. INTRODUCTION

Traditional Genetic Algorithm uses mutation operator that involves a probability that a bit of a chromosome will be changed from its original state. This type of mutation involves randomness and adds diversity to the current generation of the population. This type of mutation mimics the random mutation that occurs in the nature. More can be read about classical Genetic Algorithm here [1].

In this paper a new approach towards mutation is introduced which follows the theory of Adaptive mutation. This paper introduces how mutation is made adaptive by detecting a pattern among the high performing chromosomes and then mutating the current population according to it.

II. ADAPTIVE MUTATION

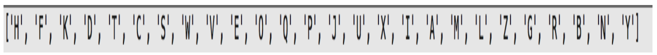
The theory of Adaptive mutation proposes that mutation may occur as a direct consequence of stress. This mutation that may occur might allow adaptation to that stress.

Adaptive mutation can be simply understood, as some phenotypic feature that may undergo some external pressure, will learn to adapt to it permanently with course of time through mutation. More can be read about biological aspect of adaptive mutation here [2].

III. THE PROBLEM USED

To test the adaptive mutation design, the famous Travelling Salesman Problem is used.

In the algorithm 26 cities are generated and a randomly generated distance is allocated between each city. The allocated distances are symmetric. The 26 cities are labeled from A-Z. Chromosome length is 26. Each chromosome represents a possible route. Each chromosome contains all 26 cities with no repetition, as shown in Figure-1 the chromosome constitutes a valid route.



```
['H', 'F', 'K', 'D', 'T', 'C', 'S', 'W', 'V', 'E', 'O', 'Q', 'P', 'J', 'U', 'X', 'I', 'A', 'M', 'L', 'Z', 'G', 'R', 'B', 'N', 'Y']
```

Figure 1. Example of a Chromosome

IV. ADAPTIVE DESIGN

To accomplish adaptive mutation a *detectPattern*() function is used which collects data of healthy chromosomes from each generations till the mutation and then detects a pattern among them. Finally when the mutation occurs it mutates the low performing chromosomes to resemble the detected pattern and hence improves the performance of the algorithm.

A. detectpattern():

This function follows the following algorithm to compute a healthy pattern.

```
Process: P1
    a.  Store fittest chromosome of each generation in a
        multidimensional array
Repeat P1 for each generation
Process: P2
    b.  If (Mutation)
            a)  Find city c that repeats itself most in ith
                column
            b)  Store c in an array of size 26 at ith
                position
Repeat P2 for i in range (0,25)
```

After completion of this algorithm a single dimension array will be obtained which will contain all the position of the cities that can produce better results.

B. Mutation Operator:

The mutation operator causes the main mutation. The *adaptiveMutation*() function is used in the code to implement the adaptive mutation operator. In this function random numbers are generated from (0,25) which select the position of cities from array such that no same city is selected.

It performs mutation by inserting cities obtained from array into the chromosomes of current generation at the same position as they were in the array. Cities in the chromosome those are same as that of the ones, which are inserted in the chromosome, are simply deleted so that a proper path is created without any repetitive city.

The mutation probability for the algorithm is set to 0.50. As the population is sorted from highest to lowest performing chromosome, so only half of the population is affected.

The mutation probability is also adaptive. It is adaptive in a sense that mutation will only affect those chromosomes that are not performing well and will leave the high performing ones unaltered. More can be read about it here [3].

V. TESTING

To test the new operator, its performance is compared to traditional mutation operator in Genetic Algorithm. The program runs normally till the point where mutation has to occur and then it splits into two parts. These parts run the instances of the algorithm, one with adaptive mutation and other one with traditional mutation. In this manner both mutations act on the same initial population. This makes the testing more efficient.

VI. IMPLEMENTATION

Code is implemented by keeping functional approach in mind. Traditional Genetic Algorithm used in my code uses following main functions.

- ***a.*** ***initialPopulationGenerator():*** Simple function that generates random initial population. The initial size of the population is set in the beginning. In this paper the initial population size is thousand chromosome.
- ***b.*** ***selection():*** It performs three functions. Firstly, it computes the fitness of each chromosome and creates a new data structure that contains all the fitness value of each chromosome. Secondly, it sorts the population from highest to lowest performing chromosome. Thirdly, it removes the low performing chromosomes.
- ***c.*** ***orderedCrossover():*** When dealing with problems like Travelling Salesman it should be kept in mind that any changes that are made in chromosome should not cause repetiton among the cities in a chromosome. To overcome this constraint this function performs ordered crossover on the current generation of population.
- ***d.*** ***traditionalMutation():*** This function performs traditional mutation. Swap mutation is used, as in travelling salesman problem we have to make sure that in the chromosome no repetition occurs as it would be an invalid path. In swap mutation two genes are randomly selected and then their positons are interchanged.
- ***e.*** ***setcities():*** This function is used to generate random cities labelled from A-Z. The cities generated are assigned random distances. The distance allocated are symmetric. In this way this function creates a virtual database of cities and their distances.

Only *initialPopulationGenerator()* and *setcities()* are called once in the beginning. All the other functions in the algorithm are inside a loop. A single iteration represents a current generation and the genetic operation that are performed on it.

There is no special fitness operator used but there is a function *chromosomeDecoder*(), which accepts a chromosome, decodes it and calculates the fitness associated with it and returns the fitness value. The fitness value is calculated simply by adding the distances between the cities hence the total distance covered in route represents the fitness of the chromosome. Smaller distances means better performance.

Algorithm goes normally till the mutation is encountered. After that the algorithm splits into two parts. In one instance adaptive mutation is applied and in other traditional mutation. Both instances of algorithms run parallel. This makes it easier to evaluate and compare the performance and also as both the mutation operators are applied on the same generation of the population, which in turn makes the testing more accurate.

VII. PERFORMANCE

Figure 2 represents the performances of both the algorithms, traditional and adaptive, over each successive generation after mutation has occurred. It can be clearly seen from the figure that the new operator performs better than the traditional operator over each successive generation.

The adaptive mutation operator not only improves the overall fitness of the population but also the low performing chromosomes are affected and perform better. As it can be seen from the figure that algorithm converges in fewer generations as compared to traditional algorithm.

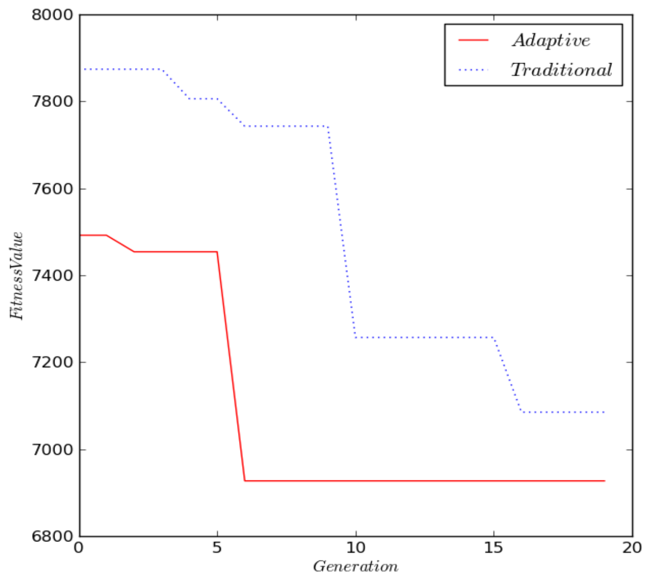


Figure 2. Maximum fitness comparison between traditional and adaptive mutation.

VIII. CONCLUSION

This paper introduced a completely new approach towards mutation in Genetic Algorithm. The adaptive approach presented in the paper opens up a lot of scope for future work on both mutation operator and way the selective pressure is implemented in algorithm. It is notable how the algorithm converges in fewer generations as compared to traditional approach. After rigorous testing and analysis of result it can be seen that algorithm can perform much better with adaptive approach towards mutation rather than just making it random as done in traditional mutation.

This approach helps to solve one of the major pitfalls of Genetic Algorithm of mutation being random. Now the change that will occur in the population is not random but controlled by the overall performance of the algorithm. A good feature of new operator is that it will only change those chromosomes that are not performing well whereas the healthy chromosomes will be left unchanged. All these features increase overall efficiency of the algorithm.

IX. ACKNOWLEDGMENT

I would like to thank my father Dr. Arun Kumar Raghava for helping me with biological aspect of adaptive mutation and Mr. Akhilesh Kumar for introducing me to Genetic Algorithm.

X. REFERENCES

[1] Goldberg DE , ”Genetic Algorithms in search optimization and Machine Learning,” Addison Wesley, 1989.

[2] Susan M. Rosenberg, ”Evolving responsively: Adaptive Mutation,” Macmillan Magazines Ltd, Volume 2, July 2001.

[3] S. Marsili Libelli, P. Alba, “Adaptive mutation in Genetic Algorithms,” Soft computing (2000) 76-80, Springer-Verlag, 2000.

[4] Grefenstette JJ, ”Optimization of control parameters for genetic algorithm,”IEEE Trans syst Man Cybern 16, 1986.

[5] Davis L, ”Handbook of Genetic Algorithms,” New York: Van Nostrand Reinhold,1990.