BULGARIAN ACADEMY OF SCIENCES
PROBLEMS OF ENGINEERING CYBERNETICS AND ROBOTICS • 2024 • Vol. **81**, pp. 13-22
p-ISSN: 2738-7356; e-ISSN: 2738-7364
https://doi.org/10.7546/PECR.81.24.02

Using DNA Analogy in Genetic Algorithms Instead of RNA Analogy

Gergana Mateeva, Dimitar Parvanov, Petar Tomov

Institute of Information and Communication Technologies
Bulgarian Academy of Sciences, Sofia, Bulgaria
gergana.mateeva@iict.bas.bg, dimitar.parvanov@iict.bas.bg, petar.tomov@iict.bas.bg

Abstract: Drawing inspiration from natural selection and genetic principles, genetic algorithms serve as heuristic global optimization tools. They excel in tackling intricate optimization and simulation challenges that conventional methods struggle to solve. While genetic algorithms traditionally align more with RNA concepts, this paper introduces a DNA-inspired adaptation of these algorithms. The study validates this modification through programmatic implementation, leveraging the openGA open-source library, and assesses its efficacy using established optimization benchmark functions. The C/C++ source code is executed on a mobile device running the postmarketOS Linux distribution.

Keywords: *chromosomes encoding, genetic algorithms, heuristic global optimization*

1. Introduction

The basic idea behind genetic algorithms is to mimic the process of natural selection and reproduction to generate a population of potential solutions to a problem. The calculation starts with a random population of candidate solutions and then applies operators such as selection, crossover, and mutation (Lambora et al., 2019) to create new generations of candidate solutions.

1.1. Selection in Genetic Algorithms

During the selection phase, the emphasis lies in favoring solutions that address the problem, a pivotal aspect within genetic algorithms. As outlined by Miller and Goldberg in 1996, this process involves choosing the most adept individuals within a population to serve as parents for the subsequent generation. The selection operator's pivotal function is to determine which individuals contribute genetic material for the next stage, significantly influencing the algorithm's efficacy and performance. Genetic algorithms employ various selection methods, each presenting distinct advantages and drawbacks. The primary ones include:

- Roulette Wheel Selection: This method apportions selection probabilities to individuals based on their fitness levels. The likelihood of an individual being chosen is directly proportional to their fitness, with the sum of probabilities across the population totaling one. Higher fitness confers a greater chance of being chosen as a parent for the next generation.
- Tournament Selection: This approach randomly picks a set number of individuals from the population, selecting the most fit among them as the parent for the subsequent generation. The tournament size is typically small, ensuring a higher likelihood of choosing the most suitable individuals.
- Rank Selection: This method ranks individuals according to their fitness, assigning the top-ranking individual a rank of one. Selection probability hinges on an individual's rank, favoring those with higher ranks.

The selection operator's significance is underscored by its role in recognizing and perpetuating the best solutions within the population, progressively enhancing the overall fitness over time. The genetic algorithm efficiently converges towards an optimal solution by opting for the fittest individuals as parents for the next generation. However, achieving a balance between selection pressure and genetic diversity is crucial to ensure a smooth convergence toward a solution that might not be optimal but is nonetheless highly effective.

1.2. Crossover in Genetic Algorithms

During the crossover phase, multiple solutions converge to form a fresh solution—a pivotal process within genetic algorithms that mirrors the reproductive mechanism seen in biological organisms. This step involves interchanging genetic material between two parental entities to generate offspring imbued with a blend of their characteristics. The primary goal of the crossover operator is to augment

the population's genetic diversity, fostering novel solutions potentially better suited for the problem at hand.

The core concept underlying the crossover operator is amalgamating genetic content from two parents to spawn one or more offspring. Typically, this operator is employed based on a probability known as the crossover rate, dictating the likelihood of performing a crossover between a given pair of parents. Genetic algorithms employ diverse crossover operators (Umbarkar & Sheth, 2015), including:

- One-Point Crossover: This operator designates a single point along the parents' chromosomes and exchanges genetic material on either side of that point, resulting in two new offspring.
- Two-Point Crossover: Two points along the parents' chromosomes are chosen to swap genetic material, generating two fresh offspring.
- Uniform Crossover: Genetic material from both parents is randomly selected and exchanged to produce two new offspring.

The selection of a specific crossover operator hinges on the problem's nature and the population's characteristics undergoing evolution. Generally, crossover amalgamates favorable traits from both parents while minimizing unfavorable ones. However, it can introduce novel genetic material that might need to exhibit superior fitness compared to the original.

A potential challenge associated with the crossover operator is premature convergence, where the algorithm swiftly converges towards a sub-optimal solution. Maintaining a balance between crossover and other genetic operators – like mutation – is crucial to address this risk. Controlling the crossover rate ensures sustained genetic diversity throughout the evolutionary process, thus preventing the algorithm from getting trapped in local optima, which are sub-optimal solutions for the problem.

Random alterations are introduced to solutions in the mutation phase, injecting diversity into the population. This element is fundamental within genetic algorithms, instigating slight, random changes to an individual's genetic makeup within a population. The mutation operator serves two primary purposes: upholding genetic diversity among individuals and preventing the algorithm from stagnating within local optima – sub-optimal solutions for the problem being addressed.

1.3. Mutation in Genetic Algorithms

The mutation operator, as described by Greenwell et al. in 1995, is activated based on a probability known as the mutation rate, determining the likelihood of a gene

within an individual undergoing mutation. Typically set at a low percentage: 1% or 5% – this rate ensures only a tiny population undergoes mutation at any given instance. Various mutation operators exist within genetic algorithms, encompassing:

- Bit Flip Mutation: Alters a single bit in an individual's chromosome representation, toggling the gene's value between 0 and 1.
- Gaussian Mutation: Introduces minute random fluctuations to a gene's value, determined by a Gaussian distribution centered around 0, often with a slight standard deviation.
- Swap Mutation: Interchanges values between two genes in an individual's chromosome, creating fresh trait combinations that might prove advantageous.

The selection of a mutation operator hinges on the problem's nature and the evolving population's characteristics. Generally, mutation sustains genetic diversity by introducing new genetic material that is potentially beneficial in problem-solving. However, excessive mutation can lower overall population fitness, necessitating a balanced use of mutation alongside other genetic operators, such as crossover. This equilibrium ensures that the genetic algorithm efficiently converges toward an optimal solution while preserving diversity.

Over successive iterations, the population progresses toward superior solutions as these operators iteratively generate new candidate solutions. Genetic algorithms find application across diverse domains, from engineering optimization to game-playing and machine learning challenges.

1.4. Biological Genetics

RNA, or ribonucleic acid, transfers genetic information within living organisms. Similarly, genetic algorithms utilize a candidate solution population as a genetic code, conveying potential problem-solving information akin to RNA's transmission from DNA to ribosomes. Like RNA, these algorithms employ selective pressure and random mutation to generate diverse candidate solution populations, assessed based on their problem-solving fitness. The progression of these populations mirrors the evolutionary process of genetic information in living organisms, where the most adept solutions endure and proliferate while weaker ones diminish. The heuristics within genetic algorithms simulate real-world evolutionary processes.

This study focuses on pairing single genetic chromosomes to mimic DNA structure, where each candidate solution possesses its complementary counterpart, creating complementary pairs.

The paper's subsequent sections are structured as follows: the second section outlines the conceptual framework, the third section details the practical implementation via a C/C++ program, the fourth section delves into the experimental aspects and their outcomes, and finally, the last section provides conclusions and suggests avenues for further research.

2. DNA Encoding

Genetic algorithms, widely recognized for their population of chromosomes (or individuals), simulate the RNA structure in biology. Although binary encoding remains prevalent, it might not suit every optimization or simulation challenge universally. Real number encoding often proves more effective for problems involving real numbers like Rastrigin, Sphere, Rosenbrock, and Styblinski-Tang functions. Real number encoding directly represents the problem domain, using chromosome genes to convey real values instead of binary ones. This approach facilitates quicker convergence and enhanced solutions, particularly for continuous variable problems.

Diverse methods exist for real number encodings, such as floating-point or Gray encoding. Floating-point encoding represents chromosome genes as floating-point numbers, while Gray encoding converts genes, portrayed as a sequence of binary digits, into real numbers via a specific formula. Extending the genetic algorithm analogy to paired DNA strands is feasible with double-stranded genetic algorithms (DSGAs). Each chromosome in the population has a complementary pair, akin to DNA base pairs. DSGAs treat chromosome pairs as single entities during selection, crossover, and mutation.

Employing paired chromosomes in DSGAs presents advantages over traditional genetic algorithms. Paired chromosomes can facilitate more efficient search space exploration by providing complementary information. They can also better handle optimization problem constraints. However, DSGAs might be more computationally demanding due to maintaining complementary pairs and executing operations on pairs rather than individual chromosomes.

An alternative proposition involves organizing chromosomes as binary twins with inverted bits, termed bit-string twin optimization. Each population chromosome consists of two twins, where one twin's bits are inverted to form the other. This approach leverages the versatility of binary encoding for any problem representation.

Twin chromosomes share a single fitness value, calculated for each twin, preventing premature convergence to sub-optimal solutions. They contribute

complementary information about the search space and maintain diversity in the population, even if one twin's fitness significantly lags.

However, employing twin chromosomes may incur additional computational costs, requiring the inversion operation for each bit. Moreover, this method may not be optimal for problems unsuitable for binary encoding.

3. Technical Implementation

The C/C++ programming languages are frequently utilized in practical simulations involving genetic algorithms, offering a direct approach to interpreting double numbers as binary bytes. In C/C++, a double variable is typically depicted as a 64-bit binary value, with the bits arranged into distinct sections representing the number's sign, exponent, and mantissa.

To manipulate the binary representation of a double variable in C/C++, one can take the address of the double variable and assign it to an unsigned long pointer. This facilitates the treatment of the bits within the double variable as unsigned integer numbers, which can be manipulated using bit-wise operators like AND, OR, XOR, and bit shifting – contrary to what was mentioned in (Yordzhev, 2012).

For instance, if there is a need to represent a binary string using a double variable “x” the following C/C++ code can accomplish this:

```
    unsigned long long *p = reinterpret_cast<unsigned long long*>(&x);
    unsigned long long bits = *p;
```

The `reinterpret_cast` operator converts the address of the “x” variable into an unsigned long pointer, storing the binary representation of “x” in the “`bits`” variable as an unsigned integer.

Once the binary representation of double variables is obtained, they can be employed as chromosomes within a genetic algorithm's population. Standard genetic algorithm operations such as selection, crossover, mutation, and evaluation can then be applied.

The openGA library, an open-source genetic algorithm library developed in C/C++ (Mohammadi et al., 2017), is a popular choice for implementing genetic algorithms. It offers an array of features enabling customization of the algorithm's behavior.

Featuring a modular architecture, the openGA library allows users to choose different components for various parts of the genetic algorithm – selection,

crossover, mutation, and fitness evaluation. This modularity fosters easy experimentation with different settings to identify the optimal configuration for a specific optimization problem.

The library encompasses genetic operators like selection, crossover, and mutation. It also supports binary and real number encoding, making it adaptable for simulations and optimizations across diverse problem domains. Its cross-platform nature ensures usability across operating systems such as Windows, Linux, and macOS, allowing users to execute genetic algorithm simulations on varied hardware platforms.

Overall, openGA enjoys popularity as a valuable genetic algorithm library for implementing optimization problems. Its modular design and support for different encoding types render it flexible and customizable, while its portability facilitates easy deployment across various hardware platforms.

For benchmark functions, real numbers serve as the encoding, but the implementation of mutation occurs using binary data. The crossover remains within the realm of real numbers. Binary twins are represented within a single array of double values, with the first half signifying the first twin and the second half denoting the second twin. Initially, binary twins are formed as bit-inverse complements. The paired complement undergoes evaluation individually, and the better fitness among both twins is considered.

4. Experiments & Results

The function proposed in (Balabanov, 2020) serves as a benchmark in experimental simulations. Its analytical form, with 'n' denoting the dimension of decision variable space and summations reaching their upper limits, is detailed.

$$f(x) = n + \sum_{i=1}^{n} x_i^2 + \sum_{i=1}^{n} \sin(2\pi x_i) \quad (1)$$

In the two-dimensional version used, the optimization region's surface resembles that depicted in Fig. 1, characterized by numerous flat regions and local optima, posing a significant challenge in locating the global optimum.

The hardware employed for these simulations is the Samsung A3 (2015) running postmarketOS (White and Qin, 2022), operating on Linux samsung-a3 6.0.2-msm8916 #1 SMP PREEMPT Sun Oct 23 21:35:55 UTC 2022 aarch64 Linux. This choice of simulation platform is informed by prior research in mobile distributed computing (Issarny et al., 2004).

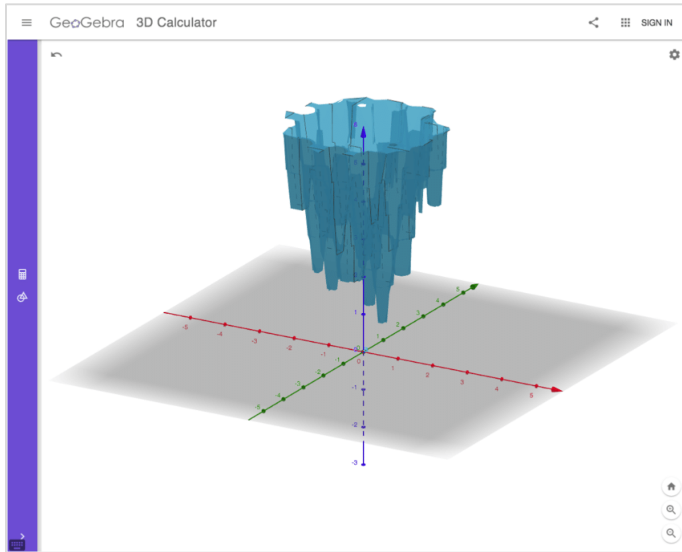

Fig. 1. A two-dimensional version

The source code for these experimental simulations is available in (Balabanov, 2023), demonstrating smooth optimization convergence, as illustrated in Fig. 2.

Despite the mobile device's comparative lower power (Satyanarayanan, 1996) compared to workstations or servers, the optimization process proves efficient, yielding solutions proximal to the global optimum. This efficiency suggests practical applicability for solving real-world problems.

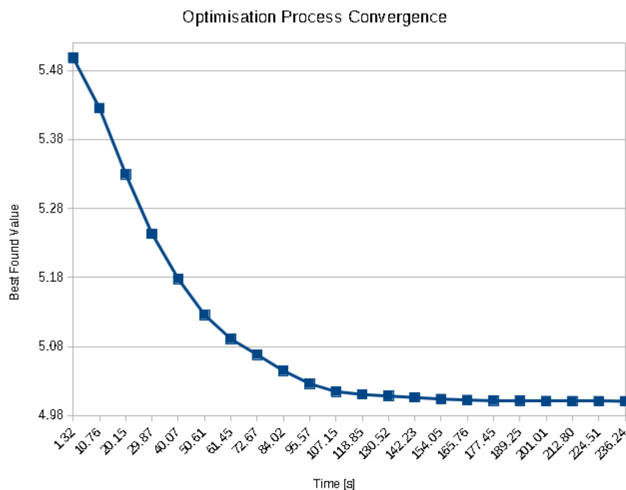

Fig. 2. Convergence process

4. Conclusion

Ultimately, adapting a DNA-inspired genetic algorithm for use on a mobile device offers the potential for enhanced problem-solving efficiency, optimizing solutions through evolutionary processes. This adaptation holds particular promise for resource-limited mobile devices, empowering them to execute complex tasks faster and more precisely. Future research could explore additional benchmark functions and examine various mobile hardware platforms to expand upon these findings.

References

1. Greenwell, R.N.; Angus, J.E.; Finck, M.: Optimal mutation probability for genetic algorithms. Mathematical and Computer Modelling, vol. 21(8), 1995, pp. 1-11.
2. Jamil. M.; Yang, X; Zepernick, H.J.: 8 - Test functions for global optimization: A comprehensive survey. Swarm Intelligence and Bio-Inspired Computation, 2013, pp. 193-222.
3. Issarny, V.; Tartanoglu, F.; Liu, J.; Sailhan, F.: Software architecture for mobile distributed computing. In: Proceedings of Fourth Working IEEE/IFIP Conference on Software Architecture, Oslo, Norway, 2004, pp. 201-210.
4. Lambora, A.; Gupta, K.; Chopra, K.: Genetic algorithm - A literature review. In: International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon), Faridabad, India, 2019, pp. 380-384.
5. Miller, B. L.; Goldberg, D. E.: Genetic algorithms, selection schemes, and the varying effects of noise. Evolutionary Computation, vol. 4(2), 1996, pp. 113-131.
6. Mohammadi, A.; Asadi, H.; Mohamed, S.; Nelson, K.; Nahavandi, S.: OpenGA, a C++ Genetic algorithm library. In IEEE International Conference on Systems, Man, and Cybernetics, Banff, AB, Canada, 2017, pp. 2051-2056.
7. Satyanarayanan, M.: Fundamental challenges in mobile computing. In Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing, 1996, pp. 1-7.
8. Umbarkar, A. J.; Sheth, P. D.: Crossover operators in genetic algorithms: a review. ICTACT journal on soft computing, vol. 6(1), 2015, pp. 1083-1092.
9. White, W.; Qin, X.: Operating system convergence: An example via the Maru OS project. In IEEE International Parallel and Distributed Processing Symposium Workshops, Lyon, France, 2022, pp. 1018-1027.
10. Yang, S.: PDGA: the primal-dual genetic algorithm. IOS Press. 2003, pp. 1-10.
11. Yordzhev, K. 2012. "An Example for the Use of Bitwise Operations in Programming". ArXiv, abs/1201.1468.
12. Zang, W.; Zhang, W.; Wang, Z.; Jiang, D.; Liu, X. and Sun, M.: A novel double-strand DNA genetic algorithm for multi-objective optimization. In IEEE Access, vol. 7, 2019, pp. 18821-18839.

13. Balabanov, T.: Global optimization benchmark function?”. ResearchGate, 2020, https://www.researchgate.net/ post/Global_optimization_benchmark_function , last visited 24 Mar 2023.
14. Balabanov, T.: DNA inspired genetic algorithm modification - C/C++ Implementation”. ResearchGate, 2023 https://www.researchgate.net/publication/369537630_DNA_Inspired_Genetic_ Algorithm_Modification_-_CC_Implementation, last visited 26 Mar 2023.
15. GeoGebra 3D Calculator. https://www.geogebra.org/3d, last visited 24 Mar 2023.
16. Samsung Galaxy A3 2015 (samsung-a3). https://wiki.postmarketos.org/wiki/Samsung_Galaxy_A3_2015_(samsung-a3), last visited 24 Mar 2023.