Artificial Immune System Algorithm Framework for Bound-Constraint Numerical Problems

Nebojsa BACANIN, Milan TUBA
Faculty of Computer Science
University Megatrend Belgrade
Bulevar umetnosti 29, N. Belgrade
SERBIA
nbacanin@megatrend.ac.rs, tuba@ieee.org

Abstract: In this paper we present our implementation of an immune system algorithm for solving unconstraint optimization problems. Proposed framework is robust and flexible, and can easily be adjusted to different optimization problems. Also, further upgrades and modification can be implemented with little effort. Framework is based on object-oriented principles and multi-tier design paradigm. We tested our algorithm on four standard optimization benchmarks. Algorithm and framework implementation are described in detail, as well as the test results with explanations and comparisons.

Key-Words: - Optimization metaheuristisc, artificial immune system, nature inspired algorithms

1 Introduction

Sometimes we face complex problems which cannot be solved in a reasonable amount of computational time. In such cases, classical optimization techniques do not achieve satisfying results because it is not possible to obtain optimum solution.

Fortunately, for solving complex problems, usually we do not need to find optimal solution, but satisfying solution in reasonable time [1]. Heuristics and metaheuristics methods have been developed to tackle such problems. Heuristics can be constructive and local search. Constructive heuristics build solution gradually from the scratch until the complete solution is generated. At the other side, local search heuristics randomly chose complete solution from the search space, and try to improve it by incremental modifications. Metaheuristics are collection of algorithms which are used for defining general heuristic methods applicable on different problems [2].

Over the last decades, bio-inspired metaheuristics have been devised for solving optimization problems. Bio-inspired metaheuristics are population based approaches and can logically be divided into two groups: evolutionary algorithms (EA) and swarm intelligence algorithms. EA are inspired by natural mechanisms such as selection, recombination (crossover) and mutation. The most prominent EA representative, which was successfully applied on wide variety of problems [3], and was analyzed in detail [4], is genetic algorithm (GA).

Swarm intelligence algorithms simulate flock of birds, school of fish, colonies of ants and bees, etc. They are composed of many homogenous components called artificial agents. Local interactions between agents are guided by simple rules, while globally agents produce complex interactions and behavior which lead whole system to the desired result. One of the pioneers of swarm intelligence algorithms is particle swarm optimization (PSO) proposed by Kennedy and Eberhart [5]. Other swarm intelligence approaches include artificial bee colony (ABC) [6] along with its upgrades [7] and hybridization with genetic operators [8], ant colony optimization (ACO) and its applications [9], cuckoo search (CS) [10] and others.

Artificial immune system (AIS) algorithm is bio-inspired, population - based local search metaheuristics. It was inspired by the characteristics and behavior of the immune system in the living organisms. First, it was used just as a tool to maintaining diversity in GA population and for handling constraints in EA [11]. One of the first attempts to solve function optimization problems directly with immune system emulation was introduced by Carlos Coello and Cruz Cortes [12]. AIS metaheuristics is used to tackle various continuous and discrete optimization problems [13], [14], [15].

This research is supported by Ministry of Science, Republic of Serbia, Project No. 44006

In this paper, we present our implementation of AIS algorithm for unconstraint optimization problems. These problems can be defined as follows:

$$min\ (or\ max)\ f(x),\ x=(x_1,x_2,x_3,\ldots x_d) \in R^n, \quad (1)$$

where $x \in S$ is a real vector with $d >= 1$ components, S is search space and $S \in R^n$. R^n is defined by lower and upper variables' bounds:

$$lb_i \le x_i \le ub_i, \quad 1 \le i \le n \quad (2)$$

This paper is organized into 5 sections. After introduction, in Section 2, brief biological background necessary for understanding AIS algorithm is given, as well as details of our AIS algorithm. Section 3 provides description of AIS object-oriented framework developed for testing purposes. Section 4 shows results of numerical experiments on four standard benchmark functions to validate integrity and robustness of our AIS implementation. Conclusion and final remarks are given in Section 5.

2 AIS Algorithm in detail

2.1 Brief biological background

The immune system is responsible for detecting and combating against pathogens. Pathogens are infectious foreign elements in the organism such as bacteria, viruses and toxins. IS consists of the two main entities [13]:

- antigen (Ag) – substance that triggers immune response and
- antibody (Ab) – molecule (lymphocytes) that is able to match and confront to Ag.

It should be noted that the immune response is specific for each antigen. This characteristics makes IS even more appealing for the implementation in the nature-inspired algorithms.

There are two types of *Lymphocytes:* B lymphocytes (B cells) and T lymphocytes (T cells). For the sake of simplicity, our AIS algorithm considers only B lymphocytes.

Lymphocyte that detects Ag and best recognize its pattern will proliferate by cloning. Some of the cloned cells will be distinguished as plasma cells, while others will be recognized as memory cells [12]. The clones are then exposed to the affinity maturation process. This process is directed towards improving the binding with the Ag.

Mutation of the clones is directly proportional to their affinity to the Ag. Clones with the highest affinity will have low mutation rates, while the lowest affinity clones will undergo high mutation rates. This mechanism of selective pressure will result in the survival of the cells with the highest affinity. Due to the random nature of mutation process, some clones could be dangerous for the organism. Such clones are disposed.

Plasma cell clones are able to generate only one type of antibodies which are relatively specific to the antigen. When the antibodies eliminate antigens, the immune system with its regulatory mechanism will dispose exceeding cells and the organism will converge to the stable state. But, the immune system is able to learn from past experience and thus, some exceeding cells will remain in the body as memory cells to ensure more efficient response to the same antigen in the future. The second encounter with the same antigen is called secondary response [12].

Described cloning and mutation processes are called clonal selection principle [16]. Our algorithm follows this principle.

2.2 Proposed algorithm

In this subsection we will present our proposed algorithm with the most important details. We must emphasize that our algorithm is adjusted for solving numerical optimization problems. Function to be optimized is considered as antigen, while the population of possible solutions is considered as population of antibodies. Steps in our algorithm are given below:

1. Create initial random uniformly distributed population of N antibodies. Assign fitness and objective function value to each antibody in the population;

repeat steps 2 - 5 until stopping criteria is met (stopping criteria is generation number and it is a control parameter)

repeat steps 2a - 2c N times

2. Clone creation and clonal selection process;

2a. each antibody is reproduced in C clones. Each clone is locally mutated by a random perturbation using the current fitness in the population and affinity maturation which will be described below. The amplitude of mutation decreases when the fitness of the original parent cell increases;

2b. for each clone calculate fitness and objective function value;

2c. perform clonal selection – the best clone (clone with the highest fitness value) replaces the original parent;

3. Calculate affinity interactions between all antibodies in the system using Euclidean distance between cells;
4. Remove antibodies whose affinity with other antibodies is below a predefined threshold;
5. Introduce randomly generated antibodies into population (diversity);
6. Sort all antibodies according to its fitness and differentiate the best one.

Each antibody (potential solution to a problem) a_i (i= *1, 2, …, N*) is a *D*-dimensional vector, where *D* is number of function parameters that should be optimized. Initial population is created, as well as, random antibodies to maintain diversity in population (see steps 1 and 5 in the pseudo–code above), using the following equation:

$$a_{ij} = lb_j + rand(0,1)*(ub_j - lb_j), \quad (3)$$

where a_{ij} is *j*-th parameter of the *i*-th antibody in the population, lb_j and ub_j are lower and upper bounds of the *j*-th parameter respectively and *rand(0,1)* is a random number uniformly distributed between 0 and 1.

For antibody *i*, we calculate fitness for function minimization problems using simple fitness function:

$$fitness_i \begin{cases} \frac{1}{1+objFun_i}, objFun_i \geq 0 \\ 1+|objFun_i|, objFun_i < 0 \end{cases}, \quad (4)$$

where $objFun_i$ is value of objective function for antibody *i*.

Each parameter of clone c_i is mutated using the following equation:

$$c_{ij} = c_{ij} + af_i * rand(0,1) \quad (5)$$

where c_{ij} is *j*-th parameter of the clone of the *i*-th antibody in the population. af_i is affinity maturation parameter calculated using Eq. 6:

$$af_i = mF_i * exp(-fitness_i^*) \quad (6)$$

where mF_i is mutation factor, and it is irreversibly proportional to the fitness of the parent antibody *i*. (see Eq. 7). This is good approximation because it is impossible to calculate affinity of the antibody *i* with the antigen (function to be optimized). In this way, we ensure that the clone that is closest to optimal solution is mutated less.

$$mF_i = \frac{1}{fitness_i}, \quad (7)$$

where $fitness_i^*$ is normalized value of the fitness of parent antibody *i*. It is calculated using the following expression:

$$fitness_i^* = \frac{fitness_i - fitness_{low}}{fitness_{high} - fitness_{low}}, \quad (8)$$

where $fitness_{low}$ is the fitness of the "weakest" antibody in the population, and $fitness_{high}$ is the fitness of the "strongest" antibody in the population.

Affinity between antibody a_i and $a_{(i+1)}$ is calculated using Equation 9.

$$affinity_{i,(i+1)} = \sqrt{\sum_{j=1}^{D} (a_{i,j} - a_{(i+1),j})^2} \quad (9)$$

3 Framework implementation

For testing and validity purposes, we developed our framework for AIS algorithm. We used C# as a programming language incorporated into .NET Framework 4.5 and Visual Studio 2010 working environment. Due to space restrictions, in this section, we will describe only the most important details of our framework.

Framework is programmed using object–oriented paradigm. Object–oriented programs are robust, scalable and flexible. Class diagram is given in Fig. 1.

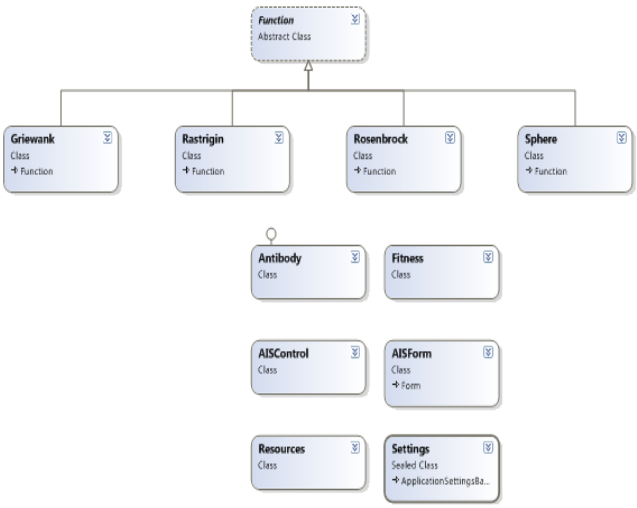

Fig 1: Class Diagram

ISBN: 978-1-61804-162-3

There are a large number of connections between classes in our program. Application is using mult-tier architecture with three underlying layers: data layer (antibody), data access layer (controller class) and user interface layer (main form). AIS algorithm cannot be used in its basic form for all function optimization problems. Unconstraint benchmark functions differ in parameter size and parameter bounds. In order to make our framework scalable and adjustable to different problems, we created *Function* abstract class which is inherited by problem specific classes. In this way, if we want to test another function, we just add another class that inherits *Function* class. Problem specific parameters are hard-coded in the application (see below). As we can see from Fig. 1, in our very first implementation, we used only four function classes that inherit *Function* abstract class.

Each antibody in the population is special instance (object) of *Antibody* class. This makes our framework slower in the sense of execution time, but more flexible to modifications and upgrades. We found this as good trade-off. *Antibody* class implements *System.ICloneable* interface which is used in the cloning process. *Antibody* constructor takes only two arguments: instance of class that inherits *Function* abstract class and instance of *Random* class with random or predefined seed. Constructor generates new antibody according to Eq. 3. Most important methods in the *Antibody* class are:

- *Clone()* – clones antibody and uses *System.ICloneable()* interface method *MemberWiseClone;*
- *calcAffinity(Antibody a)* – calculates affinity with another antibody which is passed as an argument;
- *calcNormFit (double l, double h)* – calculates normalized fitness for relevant antibody;
- *calcObjective()* – calculates objective function value for relevant antibody.

Fitness class is used to calculate fitness using Eq. 4. It is instantiated in the *Antibody* class and fitness for each antibody in the population is calculated using its only method *calcFitness().*

AISControl class is controller class that links all above described classes. All framework control parameters are implemented as global variables in *AISControl*. Also, all major loops used for iterative execution, and clone generating and selecting processes are implemented here. This class also controls sum of fitness of all population members, global affinity interactions of antibodies, sorting of antibodies according to its fitness, etc. All antibodies are stored in this class using *ArrayList* data structure and C# *Generics.* This provides obvious advantages like type-safety, performance and reusability.

Finally *AISForm* class is used to create GUI (Graphical User Interface). It takes results for *AISControl* and shows it to the user. Also, it provides user interface for parameters adjustments. Our framework provides simple and user-friendly GUI.

Framework control parameters are:

- *Na* is the number of antibodies in the initial population of antibodies;
- *Nc* defines the number of clones which are generated for each antibody;
- *Ng* is the number of generations (iterations) in algorithm`s execution (steps 2-5 in pseudo-code shown in Subsection 2.2);
- *Cst* is clonal selection threshold parameter which controls clonal selection process;
- *Rt* is remove threshold parameter. This parameter controls which antibody will be removed from the population according to its affinity with other antibodies (step 4 in pseudo-code shown in Subsection 2.2);
- *Dv* is population diversity parameter. It controls the number of new members introduced into population as a percentage of the number of current population members (step 5 in pseudo-code shown in Subsection 2.2)

Due to the nature of our framework for easy *implementation* of new functions, problem specific parameters are hard-coded. These parameters include:

- *D* is the number of function parameters;
- *Ub* is upper bound for specific parameter;
- *Lb* is lower bound for specific parameter.

4 Test results

For testing accuracy and robustness of our algorithm, we used four standard bound-constraint benchmarks. Unconstraint functions used in this test are summarized in Table 1.

All tests were performed on Intel Core2Duo T8300 mobile processor on 2.4 MHZ with 4GB of RAM memory. Windows 7 x 64 Operating System platforms was used. Code was executed in .NET Framework 4.5 environment using Visual Studio 2010 technology.

ISBN: 978-1-61804-162-3

Name	Formula	Range
Sphere	$\sum_{i=1}^{n} X_i^2$	[-100,100]
Griewank	$\sum_{i=1}^{n} \frac{x_i^2}{4000} - \prod_{i=1}^{n} cos(x_i / \sqrt{i}) + 1$	[-600,600]
Rastrigin	$10n + \sum_{i=1}^{n} (x_i^2 - 10\cos(2\pi x_i))$	[-5.12,5.12]
Rosenbrock	$\sum_{i=1}^{n-1} \left[100(x_i^2 - x_{i+1})^2 + (x_i - 1)^2\right]$	[-50,50]

Table 1: Summary of benchmark tests

Taking into account the nature of statistical tests, we ran each test 30 times consecutively with 100 generations (iterations). The average and best solutions are measured for efficiency purposed and standard deviation is used for stability comparisons. We used similar parameter set like in immune algorithms proposed for multi - objective optimization [13] [17]. Control parameter set used in our tests are showed in the Table 2.

Parameter	Value
Na	100
Nc	4
Ng	100
Cst	0.01
Rt	0.2
Dv	0.2/0.*3*

Table 2: Control parameters

As can be seen from the table, we ran tests with two different values for population diversity parameter (*Dv = 0.2* and *Dv = 0.3*). We wanted to measure the impact of this parameter on algorithm`s performance.

For evaluations, all tested function are 30 dimensional (*D = 30)*, upper (*Ub*) and lower (*Lb*) parameter bounds are specific for each test function and can be seen in Table 1. Antibodies are encoded as real values and each antibody is 30 - dimensional array which stores function parameters.

Results of proposed tests with *dv* = 0.2 and *dv = 0.3* are shown in Table 3 and Table 4 respectively.

With elevation of *dv* parameter, exploration power of the algorithm increases, because new population members are being introduced. Also, number of population members exponentially increase.

Function		Results
Sphere	Best	1.26E-5
	Mean	3.04E-4
	Stdev.	9.13E-4
Griewank	Best	2.18E-6
	Mean	3.15E-5
	Stdev.	8.41E-6
Rastrigin	Best	8.53E-6
	Mean	1.01E-5
	Stdev.	7.23E-5
Rosenbrock	Best	1.29E-1
	Mean	0.005
	Stdev.	0.003

Table 3: Tests with *dv=0.2*

By comparing Tables 4 and 5, where *dv* is set to 0.2 and 0.3 respectively, we conclude that this parameter has significant impact on algorithm`s performance. This fact is the most obvious on *Sphere*, *Rastrigin* and *Rosenbrock* tests where its bests improve noticeably (by the factor of 10^{-1}). In *Grienwank* test, only mean result improvement can be noteced.

Function		Results
Sphere	Best	0.15E-6
	Mean	6.23E-5
	Stdev.	6.09E-5
Griewank	Best	8.28E-6
	Mean	1.28E-6
	Stdev.	0.26E-6
Rastrigin	Best	5.19E-7
	Mean	9.08E-6
	Stdev.	1.15E-6
Rosenbrock	Best	0.33E-2
	Mean	0.003
	Stdev.	0.001

Table 4: Tests with *dv=0.3*

It is interesting to emphasize the impact of *dv* parameter on algorithm`s execution time. In the second case, when *dv = 0.3*, algorithm executes much slower than with *dv* set to *0.2*. Also, memory is consumed more in the second case because antibodies in the *ArrayList* (see Section 3) are dynamically added and thus more memory needs to be allocated. Low performance systems could experience *Memory Exception* error.

As we can see from Table 3 and Table 4, our AIS framework obtains satisfying results for all presented benchmarks and can be compared

ISBN: 978-1-61804-162-3

with other algorithms and software systems like the one presented in [18].

6 Conclusion

In this paper, we presented our implementation of AIS algorithm for solving unconstrained optimization problems. We developed our framework for AIS algorithm`s testing and validity purposes using object–oriented programming and multi–tier design paradigms.

The performance of the algorithm was tested on four standard unconstraint benchmark problems. We conclude that our AIS implementation has potential to handle various unimodal and multimodal problems and it is worth of further research. Further research on applying AIS algorithm on other problems similar to one proposed in [19] and [20] are worth of considering.

References:

[1] Chiong R., *Nature-Inspired Algorithms for Optimisation*, Springer, 2009, p. 536.

[2] Michalewicz Z., Fogel B. D., *How to solve it: Modern Heuristics 2nd edition*, Springer-Verglag, 2004, p. 561.

[3] Nicoara E. S., Filip F. G., Paraschiv, N., *Simulation-based Optimization Using Genetic Algorithms for Multi-objective Flexible JSSP*, Studies in Informatics and Control, Vol. 20, Issue 4, 2011, pp. 333-344.

[4] Paterlini S., Minerva T., *Genetic Algorithms in Partional Clustering: A Comparison*, WSEAS Recent Advances in Neural Netoworks, Fuzzy Systems & Evol. Computing, 2010, pp. 28-36.

[5] Kennedy J., Eberhart R., *Particle Swarm Optimization*, Proceedings of IEEE International Conference on Neural Networks, 1995, pp. 1942–1948.

[6] Karaboga D., *An idea based on honey bee swarm for numerical optimization*, Technical Report TR06, Computer Engineering, Department, Erciyes University, Turkey, 2005.

[7] Brajevic I., Tuba M., *An upgraded artificial bee colony algorithm (ABC) for constrained optimization problems*, Journal of Intelligent Manufacturing, 2012, available Springer Online First, doi: 10.1007/s10845-011-0621-6, pp. 1-12.

[8] Bacanin N., Tuba M., *Artificial Bee Colony (ABC) Algorithm for Constrained Optimization Improved with Genetic Operators*, Studies in Informatics and Control, Vol. 22, No. 2, 2012, pp. 137-146.

[9] Jovanovic, R., Tuba, M., *An ant colony optimization algorithm with improved pheromone correction strategy for the minimum weight vertex cover problem*, Applied Soft Computing, vol. 11, issue 8, 2011, pp. 5360–5366.

[10] Yang X. S., Deb S., Cuckoo search via Lévy flights, In: Proc. of World Congress on Nature & Biologically Inspired Computing (NaBIC), 2009, pp. 210-214.

[11] Kurpati A., Azarm S., *Immune Network Simulation with Multiobjective Genetic Algorithms for Multidisciplinary Design Optimization*, Engineering Optimization, Vol. 33, 2000, pp. 245-260.

[12] Coello C., Cortes C., *Solving Multiobjective Optimization Problems using an Artificial Immune System*, Genetic Programming and Evolvable Machines, Vol. 6, Issue 2, 2005, pp. 163-190.

[13] Freschi C., Repetto M., *Multiobjective Optimization by a Modified Artificial Immune System Algorithm*, Proc. of the 4th Int. Conf. of Artificial Immune Systems, 2005, pp. 248-261.

[14] Dasgupta D., Yu S., Nino F., *Recent Advances in Artificial Immune Systems: Models and Applications*, Applied Soft Computing, Vol. 11, Issue 2, 2011, pp. 1574-1587.

[15] Afshari M., Sajedi H., *A Novel Artificial Immune Algorithm for Solving the Job Shop Scheduling Problem*, Int. Journal of Computer Applications, Vol. 48, No 14, 2012, pp. 46-53.

[16] Burnet F. M., *Clonal selection and after*, Theoretical Immunology, 1978, pp. 63-85.

[17] Deb K., Pratap A., Agarwal S., Meyarivan T., *A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II*, IEEE Transactions on Evolutionary Computation, Vol. 6, 2002, 182-197.

[18] Bacanin N., Tuba M., Brajevic I., Performance of object-oriented software system for improved artificial bee colony optimization, Int. Journal of Mathematics and Computers in Simulation, Vol. 5, Issue 2, 2011, pp. 154-162.

[19] Magalhães-Mendes J., *Complex Scheduling Problems Using an Ant Optimization Methodology*, WSEAS Transactions on Information Science and Applications, Vol. 7, Issue 2, 2010, pp. 220-229.

[20] Exnar F., Machac O., *The Travelling Salesman Problem and its Application in Logistics*, WSEAS Transactions on Business and Economics, Vol. 8, Issue 4, 2011, pp. 163-173.