ISSN: 0974-5823 Vol. 7 No. 2 February, 2022

International Journal of Mechanical Engineering

Search-Based Software Code Modularization using Chaos Game Optimization

Divya Sharma*, Shikha Lohchab

Abstract

Harman and Jones [1] were the first to employ SBSE. SBSE transforms a software engineering issue into a complex search problem that a metaheuristic can solve. This primarily consists of a search space or a set of potential resolutions. SBSE (Search-based software engineering) is pertinent to nearly all segments of the software development progression. It is significant in software engineering as it offers a computerized method for addressing difficult, highly inhibited challenges for multiple purposes. Because depictions, as well as fitness functions, are readily accessible in software engineering, it is rather casual to spread over. SBSE aims to redevelop Software Engineering problems as search-based optimization issues. It should not be disorganized by textual or hypertextual penetration. It is a somewhat complex problem to solve for SBSE, one that requires a search space of candidate outcomes, driven by a fitness function which differentiates among the worst and superior options. Many different optimization and search strategies may be employed in SBSE, and this article aims to provide an overview of them. Simulated annealing, local search, genetic software design, as well as genetic algorithms are the most often employed. More advanced engineering parameters including power-driven, chemical, materials, electronic and electric engineering have been using search-based optimization for many years. Since this tendency for software engineering has just recently emerged, it has only lately been detected. Consider the possibility that this newfound interest will continue to grow in the future.

Introducing SBSE is similar to expanding an existing engineering study to include software as a vital part of procedure. On the other side, software engineering has highly upgraded and boosted up because it is indicative of search-based optimization application latent. As a result, the software is a far better candidate than outmoded engineering artifacts for search-based optimization. It is common in conventional engineering optimization to replicate the artifact being optimized. Because the item to be improved is a physical object, this is characteristically required. For search-based optimization to work, fitness evaluations must be recurrent, which is impractical if each fitness assessment requires a physical explanation. Engineers who want to include search-based optimization into conventional physical engineering outcomes must thus satisfy themselves with a technique that is one step away from reality; optimizing not the item itself, but some depiction or imitation of it. A layer of possible imprecision and expense is added to the fitness determined this way; thus, it does not reflect the final product's fitness.

Nowadays, the use of SBSE techniques has rapidly increased. Particularly in the subject of software testing, much effort has been done, and a comprehensive study of this branch has been completed [14, 18,]. Other difficulties in the software engineering field are labelled as search issues [2,4,7], which gives a quick overview of the current status of SBSE. Modularization and refactoring are well-considered as activities of "re-designing" software in this review research, which adheres to the division of software design. It has previously been disregarded in previous research of SBSE, notably in the area of short-term research in architecture near design that employs search-based methodologies. All such software engineering methods rely heavily on image and fitness functions, and this has been classified as such. While genetic algorithms [8, 9, 15] are widely used in a search-based design, the options concerning genetic operators are equally significant as well as tough to define.

Keywords: genetic algorithms, search-based software engineering, metaheuristic algorithms, software modularization, software refactoring.

Introduction

According to the current body of knowledge on how to create and maintain actual and maintainable software systems, this extends from the acknowledged research studies on software modularization [6, 9, 19], to the interpretation of object-oriented software projects [10, 12], to contemporary software plan procedures, like DDD (“Domain-Driven Design”), microservices [3-4]. Software development teams still face the challenge of devising large-scale software configurations on a regular basis. The present article discusses the difficulties of modularizing a broad software categorization. The concept of modularity is the division of a (frequently large) software structure into distinct mechanisms for minimizing the overall complexity [15]. Despite this, the process of selecting the precise modules that make up a whole software system and which classes fit into which unit is exciting. This may be much more difficult in the initial software development phases when the application itself is unknown [1,4,12]. Lastly, the way modern large software corporations work and rift the progress errands also supplement the encounter: units grow in prompt paces, are technologically advanced by different software engineering communities, and any potential junctures between modules are resolved ad-hoc. For more than a decade, investigators in the software engineering area are focusing on modularizing code. We envision the whole thing focusing on, for example, recommending code metrics which aims to detect modules that are not unified or are too connected (i.e., [16, 17, 18]), and automated modularization anti-patterns detection (i.e., [17,19]). We also observe a lot of contribution of the community in software re-modularization modeling as an optimization issue and using search-based processes to uncover potential explanations, which is closely tied to this research [3, 5, 14, 18]. Cohesiveness, number of clusters and modules, cohesion, package size, effort, number of variants, as well as semantic coherence are some of the measures used by distinct researchers to measure the quality of software. Hill climbing, NSGA-II, vanilla GA, NSGA-III, communicating GA, as well as MOEA, are some of the optimization strategies they utilize. These systems have improved in accuracy over time, but we still don't know how they'll function in large-scale corporate software systems, despite the fact that the current results are optimistic [13,19,25].

This evaluation focuses on the decisions made about the specific aspects of search algorithms; any new research study in the area of SBSE would benefit from knowing which resolutions have shown to be the most effective in the past. The following is the order in which the survey will be conducted.

Section 2 provides an introduction to simulated annealing as well as genetic algorithms through the software development lifecycle. Sections 3, 4, and 5 cover search-based architectural design, clustering, and refactoring. The background of every basic issue is identified first, followed by contemporary strategies for solving the problem using search-based methods. Furthermore, Section 6 discusses research on metaheuristic search algorithms and quality predicting models. Section 7 concludes with some suggestions for further research, and Section 8 presents findings.

SBSE in Software Development Lifecycle

SBSE for Software Testing

Many software testing systems have integrated SBSE, which includes structural, model-based, mutation, exception, temporal, integration, regression, interface, as well as configuration testing. [12, 19, 23]. Statistical testing search-based procedures have received very little attention in the literature review. Experiential evidence that the test case distributions derived by SBSE outperformed the fault-finding capabilities of standard structural testing methods. Stress testing is also discussed in research articles.

Search-Based Software for Software Enterprise

The latest focuses of SBSE research are software strategy. Software design provides the engineer with a naturally complex design space where several difficult and contradicting goals should be improved, and balances between many issues which identified [7,10].

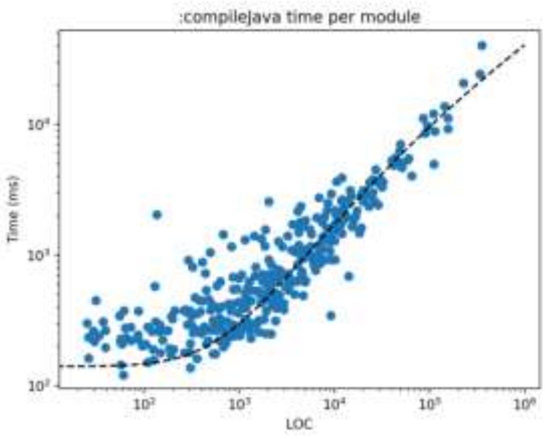


Figure 1: Runtime of compilation time and lines of code per component, and regression function

SBSE research centred on completely automated fitness addition approaches, in which human efforts into the general design of effective fitness functions is significant to SBSE's success [2]. Furthermore, there has been little prior research on the human software engineer's direct engagement in search process. SBSE [12] approaches are used by software companies to help decision-makers through the design phase. The SBSE technique may be used to find a balance between different conflicting goals using multi-objective optimization. There is a chance that these different goals may conflict. Researchers have discovered Pareto optimum approaches to be very effective in this case [23].

SBSE for Software Modeling and Prediction

SBSE is especially well suited to modeling and forecasting since it can accommodate numerous, sometimes competing objectives and constraints and may be utilized to interpose the ideal match to a collection of data employing fit as a fitness function. Using genetic programming to construct optimal models from different data sets may help increase predictive competence in software quality models. Component-based system assembly may benefit from SBSE's ability to forecast presentation characteristics [11,15].

Designing models utilizing genetic software design, a behavioural prototype is created using the SBSE methodology. The genetic programming contribution is generated by a combination of static and dynamic examinations [24,29.41].

Search Objectives

The following are the four main aims of our search:

Coupling

This goal is widely offered as a code quality method [16, 24, 28, 29], and it is usually used with search algorithms to retrieve a code base's code quality [3, 7, 10, 13, 17]). We use inter-module coupling to quantify software coupling in this study (InterMD).

InterMD estimates the total number of interdependent modules that exist between the classification's other modules. When any class from module A relies on module B, there is a dependence between the two modules A and B. As a result, the overall InterMD of a separate is equal to the sum of all module dependencies for all modules of system.

Cohesion

IntraMD coupling (intra-module) is used as a cohesion measure. Like Harman et al. [22], IntraMD develops the number of intra-module requirements (e.g., the number of dependencies among classes inside similar module) divided by maximum feasible number of intra-dependencies. Two alternative measures, based on CRP ("Common Reuse Principle") as well as CCP ("Common Closure Principle") [39], have been tested. Our early investigation revealed that employing IntraMD resulted in superior results, thus that is the one we will discuss in this work.

Number of changes

An answer's "move-class" refactoring is measured by the number of classes that are shifted to a different module in the response. We thus favour the solution with fewer modifications, given two alternatives X1 and X2 that achieve the same InterMD and IntraMD. In addition, this goal represents the time and effort designers put in to manually verify the solutions they provide (refactoring references) and then approve the modifications. However, as we have previously shown, many enhancements may be made to a large-scale codebase but the work required may not be value the quality increase, nor are organizations willing to risk significant numbers of variants at once.

Cost of unit cache breaks

As previously stated, reproducing transitive connection for our industrial partner is a vital aim. To prevent recompiling the whole code base, this module caching is employed. Building code changes may take a long time to implement, thus caching is a common solution. Modules that have been modified and their dependencies are re-compiled by the Researcher in place of re-compiling the whole software system. Since the modules are interdependent, each change has a cascading impact throughout the whole codebase. The number of caches that are broken is significantly reduced when a dependency is removed or a module is divided [14,19,24,29,34].

Using LOC ("Lines of code") of a module, FLOC forecasts the module's construction time using a linear regression model. To account for changes in actual build time, we can simply subtract the LOC of classes that have been relocated from their initial modules and add it to the modules they have been moved to using FLOC. Figure 1 illustrates the linear regression between the time and LOC of module as well as regression algorithm [20,28,37,].

Search methods

SBSE practitioners have access to a wide range of metaheuristic algorithms. Through frequent quality improvements, these technologies are utilized to automate search-based issues [11,13]. Most search-based metaheuristic algorithms are compared to haphazard search as a benchmark. Despite the non-deterministic approach utilized by most metaheuristics, a fitness function is employed to determine whether search should move forward or backward based on the validity of the initial option. This approach uses the hill-climbing approach, where the process starts from a random beginning point. After then, the fitness function is utilized to compare the two new variations in problem. New optimum solutions have developed by the algorithm based on the perceived "quality" of a given solution. The solution quality improves with time as less optimal variations are rejected and better alternatives are selected. Ultimately, an optimum or sub-optimal problem is achieved with identical functioning but a superior design. It is a quick method compared to other metaheuristics, but it has the potential to be limited to local optima like other local search methods. It is possible for the algorithm to "peak" at a less ideal option (equivalent to reaching the top of a mountain after a long climb). Search algorithms may be divided into two broad categories that vary only in one dimension [3]. The algorithm for first-ascent hill climbing is simpler, but steepest-ascent hill climbing has a somewhat more advanced search strategy and is a better option in terms of quality. Stochastic HC (neighbours are picked randomly and contrasted) and random-restart hill climbing are two further versions (the algorithm is re-initiated at various times for exploring the search space and enhance the local optima that have been found). SBSE often employs HC as a search algorithm because of the commonalities it has with other search methods. EAs that imitate the reproduction as well as mutation procedures in genetics or natural selection, such as genetic algorithms, are called EAs (evolutionary algorithms). There are many possible solutions (called "genes"), and GAs employ a "fitness function" to order them [6]. As a result, the "fittest" genes are constrained in every generation (i.e., every reiteration of the search). Each generation, a fraction of the population is chosen and utilized to breed the next generation of solutions to introduce variety into the gene pool. To generate the next generation, we must go through this selection process in two stages. To begin, the child solution(s) are generated from the parent solution(s) using a crossover operator. When using the crossover operator, choices are selected from each parent and combined to create a new child, which is determined by the algorithm itself. Mutation is the next stage once the child solutions have been formed. As previously stated, the execution of mutation is dependent on the GA issued. To keep the selection of solutions unpredictable and avoid convergence, random mutations are performed [34,39]. The newly produced solutions are reintroduced into the gene pool after mutation has been introduced to a subset of the child solutions. New solutions have been reordered based on their relative fitness to the rest of the set at this stage. Weak solutions are often eliminated from future generations by imposing a limit on the population size. Once the termination condition has been met, the procedure is repeated until it has been completed successfully. D. Multi-objective evolutionary algorithms: In software engineering, like in other disciplines, there are likely to be many competing goals that must be addressed and optimized. Instead of having to combine multiple goals into a single overarching goal, a multi-objective algorithm can take them all into account separately. MOEAs (Multi-objective evolutionary algorithms) are among the many EAs that may be utilized to solve multi-objective issues. It is possible that the additional processing required to take into account the multiple goals may increase the time required to develop a set of solutions when utilizing multi-objective algorithms for software reformation instead of MOMA ("Mono-Objective Metaheuristic Algorithms"). In addition, when an MOEA produces a population of options, the "best solution" is entirely up to the analysis of user. Where a "Single-objective EA" may rank the final population of solutions based on a single fitness value, the MOEA population can have a multitude of options depending on the importance of a particular objective fitness. The user has a variety of alternatives to choose from based on their preferences or the current scenario.

CGO (Chaos Game Optimization)

An explanation of the CGO algorithm's motivation and mathematical model is discussed in this section.

Motivation

Chaos theory is an area of mathematics that focuses on dynamical systems that are particularly sensitive to their beginning circumstances. It signifies the presence of certain main patterns in the behaviour of these systems, including repeated templates, similar loops, fractals, and multiple sub-systems, representing them as self-organized and self-similar dynamical systems, when considering the randomness of these dynamical systems. Since the future state of dynamical systems is so dependent on their beginning conditions, the chaos theory says that even slight changes to their initial conditions may have major effects. It is possible to utilize a system's existing state to forecast its future state, but the system's approximate present state does not [13,19,25].

Fractals are the most common graphical representation of chaotic processes. A fractal is a subset of Euclidian space in which a certain geometric design is repeated at different sizes. Fractals are self-similar systems because they exhibit similar forms in various sizes. A well-known fractal known as the Mandelbrot set depicts a complex infinite border in which various recursive aspects are shown at different sizes [4]. The Mandelbrot set's self-similarity is seen at various sizes in Figure 1.

To construct fractals, the chaos game involves starting with either a random point or an initial polygon form. The primary goal is to build a series of points iteratively to obtain a drawing that has a similar form regardless of the size. In this respect, the polygon vertices that is regarded to be the primary shape of the fractal need to be placed correctly. In the next step, a random initial point is chosen for the fractal. The next point in the series is decided by dividing the initial point's distance from a chosen random polygon vertex by a fraction of this distance [11,19,42].

A fractal is formed by repeating this procedure, considering the random starting point and selection of random vertex in every iteration. To construct a Sierpinski triangle, we need three vertices and a factor of 1/2. A Sierpinski Simplex having N1 dimensions may be generated by increasing the number of initial fractal vertices to N.

Sierpinski triangles may be made using the approach of chaos game [7,19,32] as an example [7,19]. The primary form of the fractal is formed by selecting three vertices, which in this example results in a triangle. Each of red, green, and blue vertices that have been chosen has been labelled. A six-sided die with two sides of red, green, and blue are rolled. In this example, the seed of the fractal is a random point that was chosen at random. Each time the dice are rolled, the initial point's seed is moved half its distance toward the related vertex, depending on which colour is revealed by the result of the roll. Using the new location of the seed, the dice are again rolled and the seed is transferred to the required vertex as a starting point for the next iteration. The final shape is the Sierpinski triangle, which is obtained by repeatedly rolling the dice. Figure 2 shows a schematic representation of the suggested technique, whereas Figure 3 depicts the final form of Sierpinski triangle and the degree to which it is self-similar at various sizes.

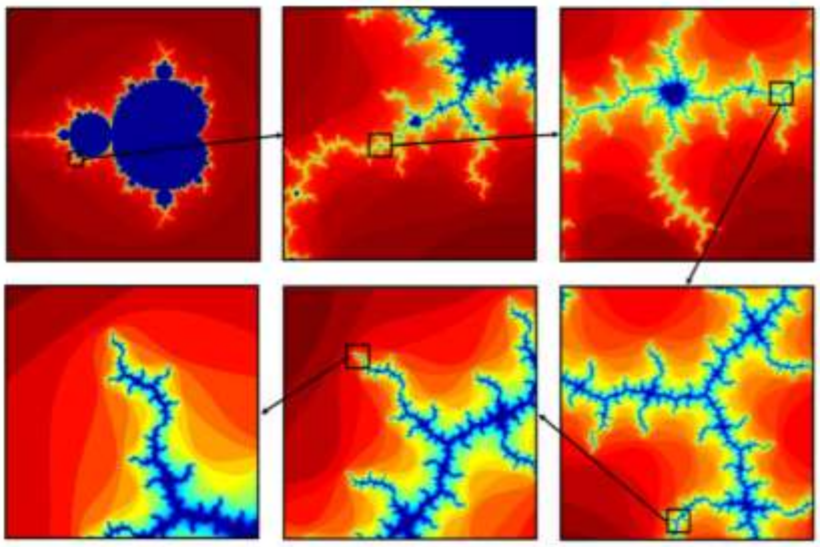


Figure 2: Proposed methodology schematic representation

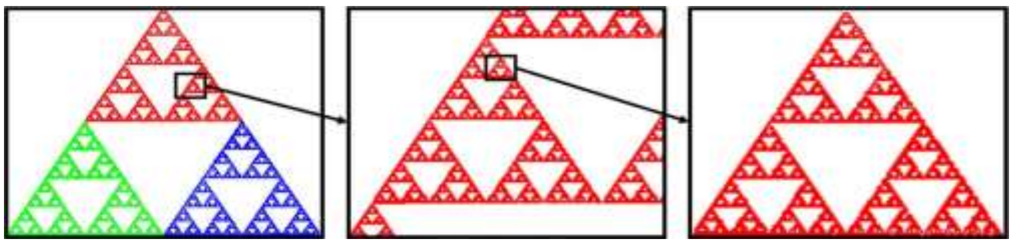


Figure 3: Self-similarity and final shape of Sierpinski triangle in different sizes

Mathematic Depiction

An algorithm based on the ideas of chaos concept is provided here. CGO algorithm's mathematical foundation is based on fractal and chaos theory concepts. Because many natural evolution algorithms retain a population of solutions that develop via random changes and selection, the CGO method analyses many solution candidates (X) that represent some suitable seeds within a Sierpinski triangle for this purpose [34,38,39].

Decision variables (xi,j) are used in this approach to represent the seed positions in a Sierpinski triangle for each solution candidate (Xi). In the optimization method, the search space for solution candidates is referred to as the Sierpinski triangle.

According to chaos theory, the behaviour of dynamic systems may be defined as self-similar and self-organized if certain key patterns in their behaviour can be identified within it. Based on the idea of chaos, the starting or (eligible seeds) reflect the fundamental patterns of dynamical systems. An optimization problem's solution candidates (X) may be used to describe the seeds' suitability as main patterns (the self-similarity). In terms of fitness values, candidates with the lowest and greatest eligibility levels are equal to the worst and best candidates [23, 28, and 32].

To make the complete form of a Sierpinski triangle, this mathematical model's central idea is to construct several suitable seeds inside the search space. The Sierpinski triangle technique of producing new seeds is also used in this context. Each seed (Xi) in search space (Xi) is represented by a temporary triangle with three seeds are as given below:

- Global Best (*GB*) position,
- Mean Group (MG_i) position,
- ith solution candidate (X_i) position as the chosen seed.

The *GB* is for the best solution candidate that has the highest eligibility levels so far, and the MG_i stands for the mean values of ceratin randomly chosen eligible seeds that have an equal chance of being included in the currently presumed to be initial eligible seed (*Xi*). The Sierpinski triangle vertices include *GB* and MG_i, as well as the picked eligible seed (*Xi*). There are two ways to complete the Sierpinski triangle: by generating a temporary triangle for each eligible seed in search area, and by producing additional eligible seeds in search area. [34,39]

This is shown schematically in Figures 4a and 4b, which provide the overall perspective of the process of constructing temporary triangles.

The primary goal of constructing temporary triangles is to generate additional seeds in search area as shown below. There are four ways to accomplish this goal. The ith temporary triangle (iteration) comprises the *n* available eligible seeds [42] from the preceding iteration, as well as three Sierpinski triangle vertices [X_i (blue), *GB* (green), and MG_i (red)]. In the chaotic game approach, three seeds and dice are used for generating additional seeds in this temporary triangle. The first seed is put in *X*, next in *GB*, whereas the third in MG_i. A dice with three red and green faces are used for first seed [43].

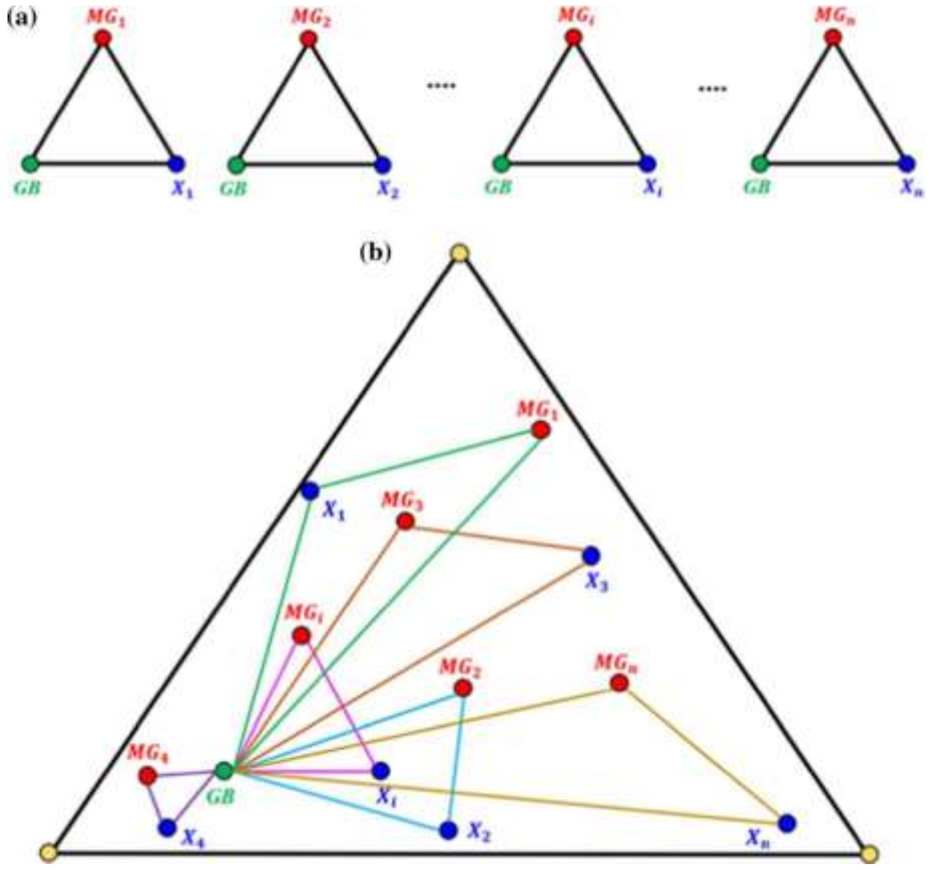


Figure 4 a. schematic representation of triangles that are temporary, b. search space with temporary triangles [35,39]

The seed in the X_i is placed toward *GB* (green face) or MG_i (red face) depending on whether the colour appears on the dice (green or red). A random integer generating function generates just two numbers, 0 and 1, providing the probability of picking red or green faces. When green face appears, seed in X_i is proceeding toward *GB*, but when red face appears, the seed in X_i is positioned toward MG_i. It is possible to generate two identical random numbers for the *GB* and MG_i even if each green or red face has an equal chance of appearing; this may be done by having one of the seeds in X_i move toward a point on one of the connecting lines between MG_i and *GB*. Because the chaotic game approach requires that the mobility of the seeds in the search area be constrained, certain randomly generated factorials are used to control this component [32,34,39].

A dice having three red and blue sides are used for the second seed (*GB*). The seed in *GB* is placed toward X_i (blue) or MG_i (red) depending on whether the colour appears on the dice (blue or red). This characteristic is modelled in the same way as the first seed was. This seed will go toward X_i if a blue face appears, but it will go toward MG_i when red face appears. The second seed, like the first seed, may travel toward a point on the connecting lines between X_i and MG_i, but this movement is restricted by certain random factorials.

Figure 5 depicts the method for the third seed in the schematic form [33,34,38,42].

Below mentioned are the highlighted location updates for mutational position changes.

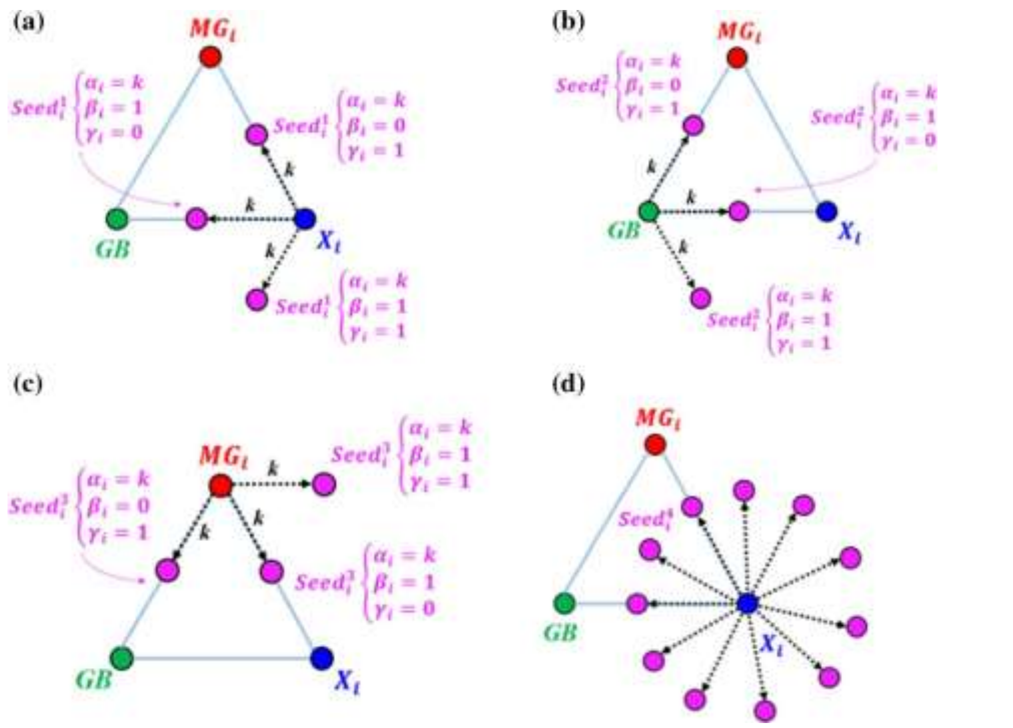

(a)
MG_i
GB
X_i
$Seed_i^1 \begin{cases}\alpha_i = k\\ \beta_i = 1\\ \gamma_i = 0\end{cases}$
$Seed_i^1 \begin{cases}\alpha_i = k\\ \beta_i = 0\\ \gamma_i = 1\end{cases}$
$Seed_i^1 \begin{cases}\alpha_i = k\\ \beta_i = 1\\ \gamma_i = 1\end{cases}$
k
(b)
MG_i
GB
X_i
$Seed_i^2 \begin{cases}\alpha_i = k\\ \beta_i = 0\\ \gamma_i = 1\end{cases}$
$Seed_i^2 \begin{cases}\alpha_i = k\\ \beta_i = 1\\ \gamma_i = 0\end{cases}$
$Seed_i^2 \begin{cases}\alpha_i = k\\ \beta_i = 1\\ \gamma_i = 1\end{cases}$
k
(c)
MG_i
GB
X_i
$Seed_i^3 \begin{cases}\alpha_i = k\\ \beta_i = 1\\ \gamma_i = 1\end{cases}$
$Seed_i^3 \begin{cases}\alpha_i = k\\ \beta_i = 0\\ \gamma_i = 1\end{cases}$
$Seed_i^3 \begin{cases}\alpha_i = k\\ \beta_i = 1\\ \gamma_i = 0\end{cases}$
k
(d)
MG_i
$Seed_i^4$
GB
X_i

Figure 5: Position update in schematic representation.

The CGO algorithm's step-by-step approach is given below, whereas the algorithm's pseudo-code is shown in Figure 6 [40,41,42].

Step 1 In search space, the initial places of solution candidates (*X*) or the first eligible seeds are determined at random.

Step 2 On the basis of self-similarity of the first eligible seeds, the fitness values of the initial solution candidates are determined.

Step 3 The seed with the greatest levels of eligibility is given the Global Best (GB).

Step 4 A Mean Group (MG_i) is calculated for every eligible seed (X_i) in search area.

Step 5 A temporary triangle having three vertices of X_i, MG_i and GB, is determined for each suitable seed (X_i) in search area.

Step 6 α_i, $þ_i$, and μ_i values are measured, for every temporary triangle.

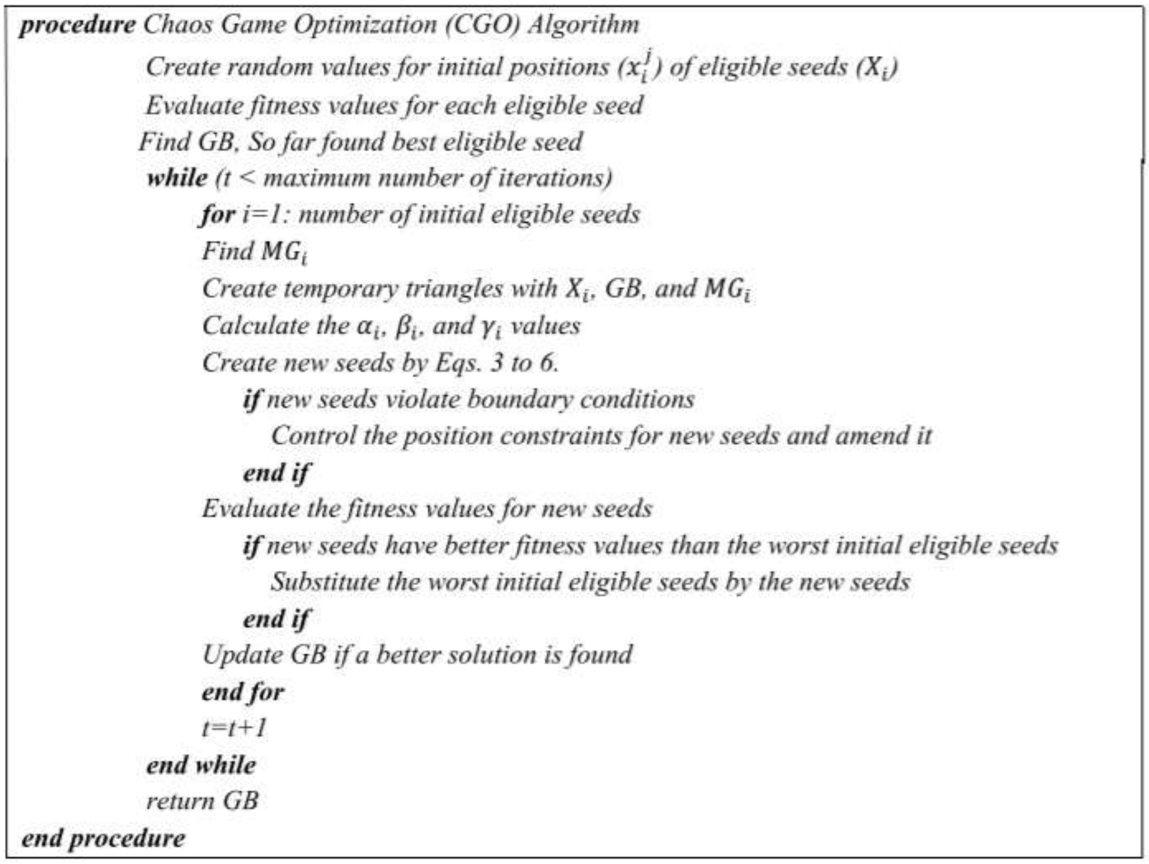
Step 7 Four seeds are prepared for each of the temporary triangles.

Step 8 A boundary condition test is performed on new seeds with ta variables outer range.

Step 9 The new seeds' fitness values are estimated on the basis of self-similarity issues.

Step 10 The new seeds replace the existing eligible seeds with the lowest fitness values corresponds to the lowest levels of self-similarity.

Step 11 Termination criteria are examined.



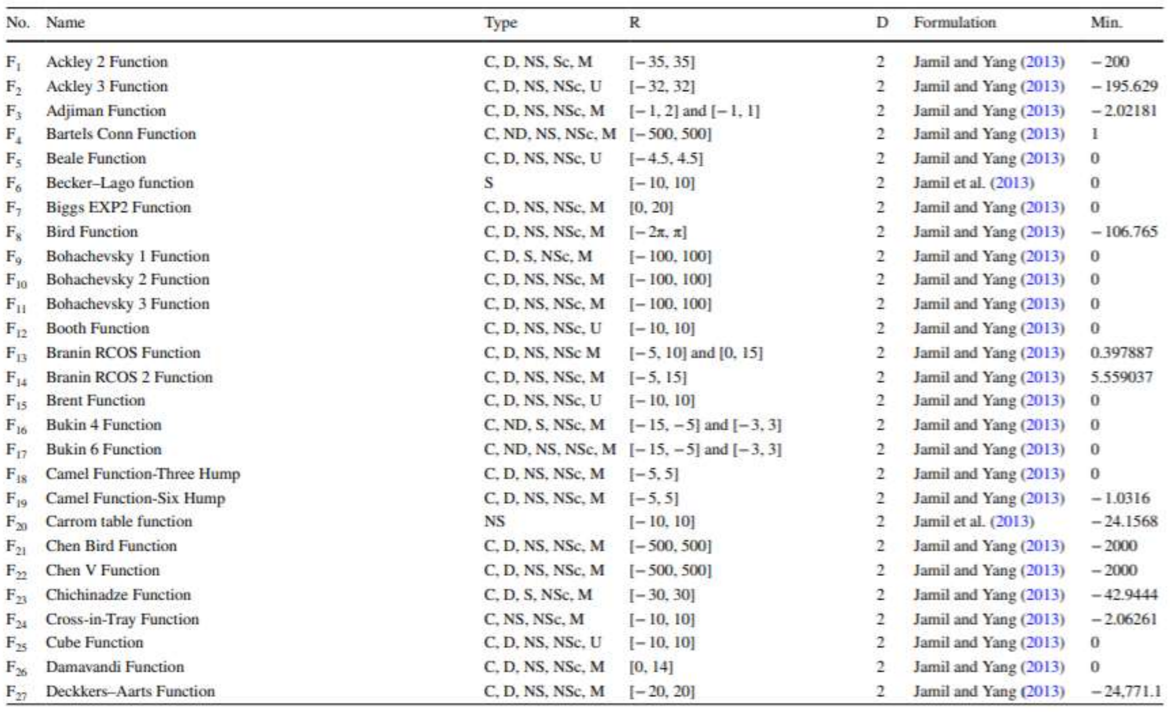
```
procedure Chaos Game Optimization (CGO) Algorithm
        Create random values for initial positions ($x_i^j$) of eligible seeds ($X_i$)
        Evaluate fitness values for each eligible seed
        Find GB, So far found best eligible seed
        while (t < maximum number of iterations)
            for i=1: number of initial eligible seeds
            Find $MG_i$
            Create temporary triangles with $X_i$, GB, and $MG_i$
            Calculate the $\alpha_i$, $\beta_i$, and $\gamma_i$ values
            Create new seeds by Eqs. 3 to 6.
                if new seeds violate boundary conditions
                    Control the position constraints for new seeds and amend it
                end if
            Evaluate the fitness values for new seeds
                if new seeds have better fitness values than the worst initial eligible seeds
                    Substitute the worst initial eligible seeds by the new seeds
                end if
            Update GB if a better solution is found
            end for
            t=t+1
        end while
        return GB
end procedure
```

Figure 6: CGO algorithm's pseudo-code [32,38,41]

Mathematical Test function

To evaluate the CGO algorithm's performance, almost every well-known mathematical function is studied in this section. As a total of 239 different types of mathematical functions, each categorized into four distinct groups, are used (Liang et al. 2005; Yang 2010a; Jamil, 2013; Yang and Jamil 2013 Yang; and Momin 2013;) where various mathematical functions are examined and produced for use in validating the current method, each with their unique properties. There are several works in optimization that make use of their method to solve a large number of problems, while just highlighting and reporting on the best solution. However, the worst outcomes and difficulties were not revealed, and the merits and demerits of algorithms remain ambiguous. An algorithm's performance should not be assessed this way. To fairly assess an algorithm's performance, we feel that a variety of issues with distinct attributes should be taken into account. Consequently, in this work, we have collected and solved practically all of the classical as well as innovative mathematical optimization functions. Furthermore, both positive and the negative findings are mentioned. Hopefully, this study ushers in a new age of research in which all findings are presented to identify all positive and negative characteristics of algorithms [40,41,42].

A total of 117 mathematical functions, with a maximum and minimum dimension of two and ten, are provided in the first group. The first 90 of these functions, known as F1 to F117, have two dimensions, whereas the last 27 have dimensions ranging from 3 to 10. Second, there are 58 test functions where the dimensions of such functions are varied because of the particular formulation, and these are referred to as N-dimensional test functions. F118 to F175, which are 50D functions, are assumed to have a maximum dimension of 50 in this context. For 3rd group, the second group's mathematical functions with a maximum 100 (100D) dimension are treated as F175 to F233. Three hybrid and composite mathematical functions, designated as F233 to F239, are taken into account for the fourth group. Table 3 lists the precise characteristics of the mathematical functions discussed in these categories. NC, C, ND, D, NS, S, NSc, Sc, M, and U signify Non-Continuous, Continuous, Non-Differentiable, Differentiable, Non-Separable, Separable, Non-Scalable, Scalable, Multimodal, as well as Unimodal, correspondingly, in this Table. Furthermore, D, R, and Min define the dimension, range, as well as the global minimum of the functions [23, 29, 41].



No.	Name	Type	R	D	Formulation	Min.
F_1	Ackley 2 Function	C, D, NS, Sc, M	$[-35, 35]$	2	Jamil and Yang (2013)	-200
F_2	Ackley 3 Function	C, D, NS, NSc, U	$[-32, 32]$	2	Jamil and Yang (2013)	-195.629
F_3	Adjiman Function	C, D, NS, NSc, M	$[-1, 2]$ and $[-1, 1]$	2	Jamil and Yang (2013)	-2.02181
F_4	Bartels Conn Function	C, ND, NS, NSc, M	$[-500, 500]$	2	Jamil and Yang (2013)	1
F_5	Beale Function	C, D, NS, NSc, U	$[-4.5, 4.5]$	2	Jamil and Yang (2013)	0
F_6	Becker–Lago function	S	$[-10, 10]$	2	Jamil et al. (2013)	0
F_7	Biggs EXP2 Function	C, D, NS, NSc, M	$[0, 20]$	2	Jamil and Yang (2013)	0
F_8	Bird Function	C, D, NS, NSc, M	$[-2\pi, \pi]$	2	Jamil and Yang (2013)	-106.765
F_9	Bohachevsky 1 Function	C, D, S, NSc, M	$[-100, 100]$	2	Jamil and Yang (2013)	0
F_{10}	Bohachevsky 2 Function	C, D, NS, NSc, M	$[-100, 100]$	2	Jamil and Yang (2013)	0
F_{11}	Bohachevsky 3 Function	C, D, NS, NSc, M	$[-100, 100]$	2	Jamil and Yang (2013)	0
F_{12}	Booth Function	C, D, NS, NSc, U	$[-10, 10]$	2	Jamil and Yang (2013)	0
F_{13}	Branin RCOS Function	C, D, NS, NSc M	$[-5, 10]$ and $[0, 15]$	2	Jamil and Yang (2013)	0.397887
F_{14}	Branin RCOS 2 Function	C, D, NS, NSc, M	$[-5, 15]$	2	Jamil and Yang (2013)	5.559037
F_{15}	Brent Function	C, D, NS, NSc, U	$[-10, 10]$	2	Jamil and Yang (2013)	0
F_{16}	Bukin 4 Function	C, ND, S, NSc, M	$[-15, -5]$ and $[-3, 3]$	2	Jamil and Yang (2013)	0
F_{17}	Bukin 6 Function	C, ND, NS, NSc, M	$[-15, -5]$ and $[-3, 3]$	2	Jamil and Yang (2013)	0
F_{18}	Camel Function-Three Hump	C, D, NS, NSc, M	$[-5, 5]$	2	Jamil and Yang (2013)	0
F_{19}	Camel Function-Six Hump	C, D, NS, NSc, M	$[-5, 5]$	2	Jamil and Yang (2013)	-1.0316
F_{20}	Carrom table function	NS	$[-10, 10]$	2	Jamil et al. (2013)	-24.1568
F_{21}	Chen Bird Function	C, D, NS, NSc, M	$[-500, 500]$	2	Jamil and Yang (2013)	-2000
F_{22}	Chen V Function	C, D, NS, NSc, M	$[-500, 500]$	2	Jamil and Yang (2013)	-2000
F_{23}	Chichinadze Function	C, D, S, NSc, M	$[-30, 30]$	2	Jamil and Yang (2013)	-42.9444
F_{24}	Cross-in-Tray Function	C, NS, NSc, M	$[-10, 10]$	2	Jamil and Yang (2013)	-2.06261
F_{25}	Cube Function	C, D, NS, NSc, U	$[-10, 10]$	2	Jamil and Yang (2013)	0
F_{26}	Damavandi Function	C, D, NS, NSc, M	$[0, 14]$	2	Jamil and Yang (2013)	0
F_{27}	Deckkers–Aarts Function	C, D, NS, NSc, M	$[-20, 20]$	2	Jamil and Yang (2013)	$-24,771.1$

Table 3: Mathematical functions representation [40,41]

Conclusion

The Chaos Game Optimization technique presented in this research is based on chaos theory ideas. Based on the chaotic game technique, the mathematical model of the algorithm is created by taking into account the possible fractal configurations and the fractals' self-similarity. To compare the new algorithm's performance against the performance of six other metaheuristic algorithms, four sets of mathematical test functions were chosen. The new algorithm's performance is evaluated using a comprehensive statistical analysis. The following are some of the most significant findings:

- While comparing other metaheuristics, the CGO approach is superior in terms of convergent to the GBs of the mathematical functions.
- According to the K-S test findings, the largest difference between CGO and other metaheuristics is often about FA.
- For the most part, the M–W test findings revealed that CGO's summing of rankings ranked lower in comparison to other metaheuristics.
- The W test revealed that CGO algorithm's mean rank scores were lower as compared to other metaheuristics in the majority of situations.
- Using K-W test, it was shown that CGO algorithm outperformed the other 2D metaheuristics in all save the number of function evaluations, whereas other metaheuristics failed to outperform it.
- According to the K-W test, CGO model has the lowest values of minimum, mean, as well as function evaluation for 50D test functions, whereas the SOS has the lowest values of standard deviation. In terms of function evaluation and standard deviation, SOS and WOA outperform CGO for 100D functions.

Because algorithm will require to be tested in the future for its ability to deal with challenging situations, numerous applications of this method might be suggested for future difficulties. In addition, novel configurations of the algorithm might be explored on the basis that researchers can have differing perspectives regarding the CGO approach described.

References

[1] D. L. Parnas, “On the criteria to be used in decomposing systems into modules,” in Pioneers and Their Contributions to Software Engineering. Springer, 1972, pp. 479–498.

[2] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Connallen, and K. A. Houston, “Object-oriented analysis and design with applications, third edition,” ACM SIGSOFT Software Engineering Notes, vol. 33, no. 5, pp. 29–29, Aug. 2008.

[3] E. Evans, Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional, 2004.

[4] S. Newman, building microservices: designing fine-grained systems.” O’Reilly Media, Inc.”, 2015.

[5] G. J. Myers, “Composite/structured design,” Google Scholar Digital Library, 1978.

[6] E. B. Allen, T. M. Khoshgoftaar, and Y. Chen, “Measuring coupling and cohesion of software modules: an information-theory approach,” in Proceedings Seventh International Software Metrics Symposium. IEEE Comput. Soc, 2001.

[7] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner, “Using automatic clustering to produce high-level system organizations of source code,” in Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242). IEEE Comput. Soc, 1998

[8] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto, “Using structural and semantic measures to improve software modularization,” Empirical Software Engineering, vol. 18, no. 5, pp. 901–932, Sep. 2012.

[9] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, “Identifying and quantifying architectural debt,” in 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). IEEE, 2016, pp. 488–498.

[10] C. K. Kwong, L. F. Mu, J. F. Tang, and X. G. Luo, “Optimization of software components selection for component-based software system development,” Computers & Industrial Engineering, vol. 58, no. 4, pp. 618–624, May 2010.

[11] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, “Design defects detection and correction by example,” in 2011 IEEE 19th International Conference on Program Comprehension. IEEE, Jun. 2011.

[12] B. S. Mitchell and S. Mancoridis, “On the automatic modularization of software systems using the bunch tool,” IEEE Transactions on Software Engineering, vol. 32, no. 3, pp. 193–208, Mar. 2006.

[13] K. Praditwong, M. Harman, and X. Yao, “Software module clustering as a multi-objective search problem,” IEEE Transactions on Software Engineering, vol. 37, no. 2, pp. 264–282, Mar. 2011.

[14] M. W. Mkaouer, M. Kessentini, M. O. Cinn ´ eide, S. Hayashi, and K. Deb, ´ “A robust multi-objective approach to balance severity and importance of refactoring opportunities,” Empirical Software Engineering, vol. 22, no. 2, pp. 894–927, Mar. 2016.

[15] H. Abdeen, H. Sahraoui, O. Shata, N. Anquetil, and S. Ducasse, “Towards automatically improving package structure while respecting original design decisions,” in 2013 20th Working Conference on Reverse Engineering (WCRE). IEEE, Oct. 2013.

[16] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, “Search based refactoring: Towards semantics preservation,” in 2012 28th IEEE International Conference on Software Maintenance (ICSM). IEEE, Sep. 2012.

[17] A. Ghanem, M. Kessentini, M. S. Hamdi, and G. E. Boussaidi, “Model refactoring by example: A multi-objective search based software engineering approach,” Journal of Software: Evolution and Process, vol. 30, no. 4, p. e1916, Nov. 2017.

[18] R. Mahouachi, “Search-based cost-effective software modularization,” Journal of Computer Science and Technology, vol. 33, no. 6, pp. 1320– 1336, Nov. 2018.

[19] A. Fadhel, M. Kessentini, P. Langer, and M. Wimmer, “Search-based detection of high-level model changes,” in 2012 28th IEEE International Conference on Software Maintenance (ICSM). IEEE, Sep. 2012.

[20] G. Bavota, F. Carnevale, A. D. Lucia, M. D. Penta, and R. Oliveto, “Putting the developer in the loop: An interactive GA for software remodularization,” in Search Based Software Engineering. Springer Berlin Heidelberg, 2012, pp. 75–89.

[21] V. Alizadeh, M. Kessentini, W. Mkaouer, M. Ocinneide, A. Ouni, and Y. Cai, “An interactive and dynamic search-based approach to software refactoring recommendations,” IEEE Transactions on Software Engineering, pp. 1–1, 2018.

[22] M. Harman, R. Hierons, and M. Proctor, “A new representation and crossover operator for search-based optimization of software modularization,” ser. GECCO’02. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002, p. 1351–1358.

[23] D. F. D. Silva, L. F. Okada, T. E. Colanzi, and W. K. G. Assunc¸ao, ˜ “Enhancing search-based product line design with crossover operators,” in Proceedings of the 2020 Genetic and Evolutionary Computation Conference. ACM, Jun. 2020.

[24] S. A. Ebad and M. Ahmed, “Software packaging approaches —a comparison framework,” in Software Architecture. Springer Berlin Heidelberg, 2011, pp. 438–446.

[25] C. Jermaine, “Computing program modularizations using the k-cut method,” in Sixth Working Conference on Reverse Engineering (Cat. No.PR00303). IEEE Comput. Soc, 1999. [26] J. K. Lee, S. J. Jung, S. D. Kim, W. H. Jang, and D. H. Ham, “Component identification method with coupling and cohesion,” in Proceedings Eighth Asia-Pacific Software Engineering Conference. IEEE Comput. Soc, 2002.

[27] Azizi M, Ejlali RG, Ghasemi SAM, Talatahari S (2019a) Upgraded whale optimization algorithm for fuzzy logic-based vibration control of nonlinear steel structure. Eng Struct 192:53–70. https://doi. org/10.1016/j.engstruct.2019.05.007

[28] Azizi M, Ghasemi SAM, Ejlali RG, Talatahari S (2019b) Optimal tuning of fuzzy parameters for structural motion control using multiverse optimizer. Struct Des Tall Spec Build 28(13):e1652. https://doi. org/10.1002/tal.1652

[29]Azizi M, Ghasemi SAM, Ejlali RG, Talatahari S (2020) Optimum design of fuzzy controller using hybrid ant lion optimizer and Jaya algorithm. Artif Intell Rev 53(3):1553–1584. https://doi.org/10.1007/s1046 2-019-09713-8

[30] Basturk B (2006) An artificial bee colony (ABC) algorithm for numeric function optimization. In: IEEE swarm intelligence symposium, Indianapolis, IN, USA, 2006

[31] Beyer HG, Schwefel HP (2002) Evolution strategies—a comprehensive introduction. Nat Comput 1(1):3–52 Cheng MY, Prayogo D (2014) Symbiotic organisms search a new metaheuristic optimization algorithm.

[32] Chu SC, Tsai PW, Pan JS (2006) Cat swarm optimization. In: Pacific Rim international conference on artificial intelligence. Springer, Berlin, pp. 854–858

[33] Dorigo M, Maniezzo V, Colorni A (1996) Ant system: optimization by a colony of cooperating agents.

[34] IEEE Trans Syst Man Cybern B Cybern 26(1):29–41

[35] Du H, Wu X, Zhuang J (2006) Small-world optimization algorithm for function optimization. In: International conference on natural computation. Springer, Berlin, pp 264–273

[36] Eberhart R, Kennedy J (1995) A new optimizer using particle swarm theory. In: MHS'95. Proceedings of the sixth international symposium on micro machine and human science. IEEE, pp 39–43

[37] Erol OK, Eksin I (2006) A new optimization method: big bang–big crunch. Adv Eng Softw 37(2):106–111 Formato RA (2007) Central force optimization. Prog Electromagn Res 77:425–491

[38] Gandomi AH, Alavi AH (2012) Krill herd: a new bio-inspired optimization algorithm. Commun Nonlinear Sci Numer Simul 17(12):4831–4845

[39] Gandomi AH, Yang XS, Talatahari S, Alavi AH (2013a) Firefly algorithm with chaos. Commun Nonlinear Sci Numer Simul 18(1):89–98

[40] Gandomi AH, Yun GJ, Yang XS, Talatahari S (2013b) Chaos-enhanced accelerated particle swarm optimization. Commun Nonlinear Sci Numer Simul 18(2):327–340

[41] García S, Molina D, Lozano M, Herrera F (2009) A study on the use of non-parametric tests for analysing the evolutionary algorithms' behaviour: a case study on the CEC'2005 special session on real parameter optimization. J Heuristics 15(6):617

[42] Geem ZW, Kim JH, Loganathan GV (2001) A new heuristic optimization algorithm: harmony search. Simulation 76(2):60–68

[43] Glover F (1986) Future paths for integer programming and links to artificial intelligence. Comput Oper Res 13(5):533–549

[44] Hansen N, Müller SD, Koumoutsakos P (2003) Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES). Evol Comput 11(1):1–18

[45] Hatamlou A (2013) Blackhole: A new heuristic optimization approach for data clustering. Inf Sci 222:175–184