Evolutionary Computation Algorithms in Automatic Test Case Generation

Liam McDevitt
Computer Science
Brock University
St. Catharines, Canada
lm15ue@brocku.ca

***Abstract*—Software testing is one of the essential tasks of a software development project. Testing takes time, money, and a vast amount of resources. We must present our program with data and see if it matches our expected output. Test cases are often generated to ensure comprehensive testing of software. Once again, developing these test cases takes time, money, and resources. However, in recent years the ability to automatically generate such test cases has seen activity within the research community. Having the ability to generate a full test suite of cases is a precious asset for any software project. Evolutionary Computation (EC) algorithms like Genetic Algorithm (GA), Bee Colony Algorithm (BCA), Ant Colony Optimization (ACO), Firefly Algorithm (FA), and Particle Swarm Optimization (PSO), along with many variations and improvements, have been used to generate suitable test cases for software in the past. This paper will explore various EC algorithms to automatically develop valuable test cases and discuss their proposed methods in detail. In addition, each presented EC algorithm will be explained at its fundamental level. It was found that PSO and GA are the two most popular EC methods for this problem, while the others have very little research backing their existence. However, this leaves room for more research on these methods in the future to add performance improvements and new ideas into the mix, hopefully.**

***Index Terms*—Evolutionary Computation, Automatic Test Case Generation, Particle Swarm Optimization, Genetic Algorithms, Ant Colony Optimization, Cuckoo Search, Bee Colony Optimization, Firefly Algorithm**

I. Introduction

It is critical to our everyday life that software lives up to a certain standard of quality since almost everything we do involves software functioning as intended daily—one of the most important, timely, and costly tasks in software testing. The laborious and expensive task of software testing can account for around 50% of software development costs [1], [2]. Testing usually comes with given inputs and expected output. The ultimate goal in software testing is to ensure the software meets the client's requirements. Test cases are often created to provide a wide range of extensive testing of software. Creating these test cases usually takes time, money and are often prone to errors themselves. Having an automatic intelligent way to generate a test suite full of cases would be a valuable asset to assure software reliability and confidence. This predicament is how the idea of automated test case generation came to be.

Various Evolutionary Computation (EC) algorithms have been used to automatically generate profitable test cases for software in the past:

- Particle Swarm Optimization (PSO) [1],
- Genetic Algorithms (GA) [3],
- Ant Colony Optimization (ACO) [4],
- Cuckoo Search (CS) [2],
- Bee Colony Optimization (BCO) [2], and
- Firefly Algorithm (FA) [5].

Many of these evolutionary techniques for automatically generating test cases are accompanied by either another tool [1], [3] or another evolutionary algorithm [2].

EC is a group of algorithms inspired by nature to help find favourable solutions to global optimization problems through the fundamental idea of biological evolution [6]. The overarching technique is to evolve a population, where each member of the population represents a candidate solution. A fitness function evaluates these candidate solutions. Over a series of generations, a population is contingent on various stochastic altercations to the candidates, i.e., selection and mutation. As the population moves from one generation to the next, our goal is to optimize our designed fitness function.

The ordinary underlying notion for test data generation using EC is that we have a search space resembling a test suite. Our fitness function is known as the "test adequacy criterion" [6]. The goal is that our test suite, from our generated inputs, meets our goal as closely as possible. Our fitness function is our main input for defining how well we're meeting our goal numerically. As will be seen from the presented papers in the later sections of this paper, there are many different types of testing where generating test data and generating test cases go hand in hand.

For software testing, there are two main methods, white-box testing and black-box testing. White-box testing focuses on finding an input that satisfies our test adequacy criterion for generating test cases, making it our primary focus. On the other hand, black-box testing is primarily used on large systems paying attention to what output you get from given inputs. For white-box testing, input data is generated and ran on multiple selected system paths [3]. An adequacy criterion for these paths can be the coverage of these paths by the generated test cases. We want to ensure we have as much path coverage of our system as possible because the more path

coverage we have with our generated test cases, the more of our program we're able to test [1]. EC algorithms have been applied to successfully generate test cases due to their global solid search ability compared to traditional methods like a unit test where an execution path is selected. An acceptable generated input is fed to the program. As the complexity rises in the structure of our approach chosen, developing sufficient program input is not trivial. Hence, we're not able to efficiently test our program, reducing the quality of our software [3], which makes EC algorithms a prime candidate for solving these kinds of problems.

The rest of this paper will be presented as follows. **Section II** presents the needed background information for this paper. All the used EC algorithms will be explained from a vanilla standpoint to help the reader gain insight into the algorithm's abilities and see how they can be incorporated into automatic test case generation methods. The explored algorithms are PSO, GA, ACO, CS, BCA, and FA. **Sections III-VII** each present a different EC-based method for automatically generating test cases. A PSO-based method [1], a GA-based method [3], a ACO-based method [4], a hybrid CS and BCA based method [2], and a FA-based method [5] will be explored throughout these sections. Each section presents a specific paper that will also discuss the methods and results of the proposed method, respectively.

II. Motivation

A. Objectives

This paper explores and explains various EC algorithms that have been used in automatic test case generation methods. Instead of diving into one specific method, we will be looking at one method per EC algorithm to see their uses and the different combination of methods presented by the scientific community to hopefully see if EC algorithms provide any significant uses in automatic test case generation or if we should look to other methods.

The work of this paper attempts to adhere to the following objectives:

- Determine any benefit of using EC algorithms for automatic test case generation.
- Recommend a preferred EC algorithm method for automatic test case generation from the ones presented.

B. Contributions

The majority of the contributions for this paper will come from analyzing work that has already being done on this topic through reviewing multiple papers.

The following points represent the contributions of this work:

- Present the reader with a fundamental understanding of various vanilla EC algorithms.
- Explore numerous variations of possible EC-inspired methods for automatic test case generation.

III. Background & Related Work

All Evolutionary Computation (EC) algorithms that will be used in this paper to generate test cases automatically will be fundamentally presented in this section. The reader must gain insight into these fundamental EC algorithms since they are the backbone of the researched method and the main focus of this paper. The topics covered include Particle Swarm Optimization (PSO), Genetic Algorithms (GA), Any Colony Optimization (ACO), Cuckoo Search (CS), Bee Colony Algorithm (BCA), and Firefly Algorithm (FA). The in-depth information into each of these algorithms will be weighed by (1) the difficulty of the algorithm and (2) it's relevance to the studied methods.

A. Particle Swarm Optimization

Kennedy and Eberhart first introduced Particle Swarm Optimization in 1995 after observing the flocking patterns of birds [7]. PSO is a stochastic, population-based algorithm influenced by the social behaviour of animals in nature. How these social entities communicate leads them towards a common goal. From a mathematical mindset, this goal is essentially an optimization problem. PSO provides excellent support for situations where there is incomplete or little to no information, given its meta-heuristic quality.

The swarm part of PSO can be thought of as a social community. A social community is filled with individuals, and in PSO's case, these individuals are particles. Particles move through an n dimensional search space where each dimension of our particle's position corresponds to a dimensional component in the overall solution. Since the position of a particle is a possible solution, is it known as a candidate solution. Over a set number of iterations, t particles in a swarm will explore the search space, trying to optimize some objective, i.e., minimize or maximize. How well a particle is doing concerning this objective is dictated by a fitness function. A particle's position improves by keeping track of its position relative to its best-known one so far and the swarm. Moving closer to the swarm is based on a topological framework of communication. There are many different ways particles can communicate in a swarm, but the most common is the global best, where all particles communicate with one another.

All particles are responsible for keeping track of three important characteristics that allow them to move through the search space. First, its current position $\vec{x}(t)$. Second, is its current velocity $\vec{v}(t)$. Finally, is its personal best position $\vec{y}(t)$. Three components go into a particle's movements through the search space: inertia, cognitive, and social [8]. The inertia component ω is a weight attached to its velocity simulating inertia influencing the particle to continue travelling in the direction it was previously going [9]. The cognitive component is how much a particle will move towards its best-known position found so far influenced by the weighted coefficient c. The social component is how much the particle will move towards the global best-found position in the swarm, weighted by s. Attached to the cognitive and social components is a uniformly distributed random vector $\vec{r}$ between 0 and 1 acting as a stochastic driving element.

Particle velocity calculation and update [10]:

$$\vec{v_i}(t+1) = \omega\vec{v_i}(t) + c\vec{r_1}(\vec{y}(t) - \vec{x_i}(t)) + s\vec{r_2}(\vec{\hat{y}} - \vec{x_i}(t)) \quad (1)$$

where $\vec{v_i}(t+1)$ is the new velocity for particle i.

Particle position calculation and update [10]:

$$\vec{x_i}(t+1) = \vec{x_i} + \vec{v_i}(t+1) \quad (2)$$

where $\vec{x_i}(t+1)$ is the new position for particle i.

B. Genetic Algorithms

A Genetic Algorithm (GA) is an Evolutionary Algorithm (EA) that is used to find favourable solutions to complex optimization problems [11]. GAs evolve a population of candidate solutions over a set number of generations. Each candidate solution is a possible solution to our problem, usually represented by a binary string, but many different representations are also possible. A fitness function is designed to indicate how well a candidate solution performs numerically. As a prerequisite to using a genetic algorithm, we need first to develop a genetic representation meaning a possible solution representation. Second, we will need a fitness function to evaluate this designed representation of a possible solution to assess individuals in the population. These two first initial steps are often the most difficult for GAs since most models are problem-dependent.

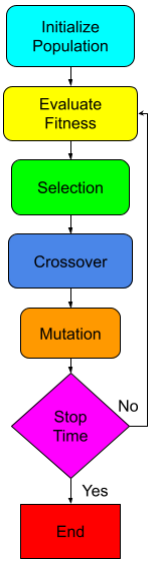

Fig. 1. Genetic Algorithm model [11].

Fig. 1 represents the fundamental flow chart of a genetic algorithm. To begin, we randomly initialize a population with possible solutions (individuals) within our search space. Once we have our population, we need to evaluate each individual's fitness based on our predefined fitness function. Next comes selection, which involves selecting parents from the population to mate. Once we've selected our parents, we do crossover, which is creating a child based on the parents. When the child is created, it then has a chance to mutate, which is often a tiny altercation in its solution. For example, if your representation of an individual were a bit string, then a possible mutation would be to flip one of the bits. In doing this process, we create new individuals for our population. We do this for however many generations until some stopping criteria are met. Usually, this criterion is based on a predefined number of generations beforehand or solved our problem. When we end the evolutionary process, the best-fit individual in the population is our answer to the problem.

C. Ant Colony Optimization

Ant Colony Optimization (ACO) is a stochastic optimization strategy mainly used to find favourable paths through graphs [12] and have been used in graph optimization problems internet and vehicle routing. The real-world behaviour of ants influences ACO due to their incredible teamwork abilities. Ants can find optimal paths through graphs by simulating their pheromone-based link. As ants traverse a graph, they leave down a pheromone trail. Pheromone from other ants will influence the movement of future ants by some probability. Along with pheromone buildup, there is also pheromone evaporation. The evaporation of pheromones over time helps to avoid premature convergence on a local optimum solution. Overall, the ants follow a positive feedback strategy to find the shortest path through a graph.

Fig. 2. Ants searching for food (Liam McDevitt, 2021).

Fig. 2 is an example of ants trying to find the shortest path from their nest to a food source. When exploring the search space, ants need to decide on which direction to go initially. The initial choice is random for all ants, and we assume they all move at the same speed. On average, if there are three possible paths, then 1/3 of the ants would try each path. Since the middle way is shorter than both the top and the bottom course, on average, more ants will take it, and hence pheromone builds quickly. For the paths less travelled, the pheromone will dissipate over time since fewer ants will cross it. The accumulation of pheromone influences the decision of new ants travelling from the nest based on a percentage affecting the probability they will choose one path over another. At this point, new ants will choose the middle path based on increased likelihood from the build-up of pheromone. Over time this increases the positive feedback affecting the ants to choose the shorter way more often until all ants choose the fastest route.

D. Cuckoo Search

In 2009 Yang and Deb introduced a new population-based meta-heuristic algorithm for solving optimization problems

called Cuckoo Search (CS) [13]. The CS algorithm was inspired by a combination of the Lévy flight behaviour of fruit flies and some birds and some cuckoo species obligate brood parasitic behaviour in nature. This inspired behaviour was simplified and will be seen through an explanation of the algorithm.

Each egg within a nest represents a candidate solution. Optimization is achieved by replacing these eggs with new cuckoo eggs to find better answers over time. The most trivial representation says that each nest will only have one egg. To effectively describe CS, the writers designed three rules: (1) cuckoo's lay only a single egg at a time and drop it off at a random nest; (2) the best nests, meaning they have the fittest eggs, get carried onto the next evolutionary generation; (3) there is a fixed number of possible host nests and a cuckoo's egg can be discovered by the host bird with a probability $p_a \in [0, 1]$. There are two options for a host bird if it discovers the egg is not it's own: (1) remove it from the nest or (2) leave its current nest to construct a new one. A trivial way for the host bird to decide is to make an approximation based on the fraction of p_a and the number of nests n.

To generate a new solution $x^{(t+1)}$ for a cuckoo i we perform a Lévy flight,

$$x_i^{(t+1)} = x_i^{(t)} + \alpha \oplus \text{Lévy}(\lambda), \tag{3}$$

where $\alpha > 0$ is the step size and should be scaled depending on the problem, but in most cases, $\alpha = 1$ is used. Entry-wise multiplications are represented by $\oplus$ and claimed to be more efficient at exploring the search space because more significant steps are taken via the Lévy flight. In essence, equation 3 is the stochastic random walk equation where the following location only depends on the current. The transition is decided by some probability, much like a Markov chain.

E. Bee Colony Optimization

The somewhat recent Bee Colony Optimization (BCO) meta-heuristic by Teodorović began a new direction for the field of Swarm Intelligence and since then has been successfully applied to complex problems like transportation [14]. The BCO algorithm was inspired by the way bees interact and communicate in nature. The core of BCO is to generate an artificial multi-agent colony of bees to solve challenging combinatorial optimization problems accurately. Of course, the algorithm does not perfectly resemble bees in nature but is influenced by their success.

The complicated task bees are best at (which we can take inspiration from) is their success at processing and acquiring nectar. Individual bees fly already to found nectar sources from other bees in their hive. All hives have what is known as a dance floor area where bees who have found a nectar source come to inform the other bees in the hive with the hopes that they will follow them to the source. If the dancing bee was lucky enough to convince another to go with them, they went together to the nectar source, where they could both bring nectar back to the hive. Once a bee returns to the hive, they have one of three choices to make based on some probability: (1) Forget about the newly found food source and go back to being an uncommitted follower. (2) Keep going back to the same food source they have found for nectar. (3) Bust a move on the dance floor, trying to recruit other bees to join them back at the food source to acquire more nectar together. How bees decide whether or not to follow a specific dancing bee is not yet understood, but the process is often seen as recruitment depending on the quality of the food source.

The BCO algorithm is a population-based algorithm, much like most EC algorithms. An individual artificial bee represents a candidate solution to our objective. The basis of the BCO algorithm is made up of two phases: (1) a forwards pass; (2) a backward pass. For a forward pass, all bees explore the search space. A new solution for the bee is generated/improved during this phase by a predefined number of moves it can analyze. Once a bee is done exploring, it heads back to the hive with its (hopefully) upgraded solution and communicates its quality with the other bees called the backward pass. Before alternating back to the first phase, there was a need to make one of three decisions based on probability. The two steps go back and forth for a certain number of generations or until some predefined stopping criteria is met.

F. Firefly Algorithm

The nature-inspired Firefly Algorithm (FA) was created in 2007 by Yang [15]. FA was based on the flashing behaviour of fireflies in nature. Fireflies can communicate with other fireflies over very long distances with their flashing lights. Other fireflies can observe both the intensity of the flash and its pattern. These flashing lights can formulate an objective function in which we can optimize. This realization is the foundation for the intuitive idea of the FA.

The FA follows three general rules to capture their flashing characteristics: (1) Fireflies are unisex, meaning any firefly can be attracted to any other not based on their sex. (2) Fireflies are attracted based on their light intensity. The less bright one will move towards the brighter one; if there's no brighter firefly, it will move about randomly. The distance between fireflies increases their brightness, and therefore their attractiveness decreases. (3) The landscape of an objective function influences a firefly's brightness.

A firefly i moves towards a brighter firefly j at time step t by,

$$x_i^{(t+1)} = x_i + \beta_0 e^{-\gamma r_{ij}^2}(x_j^{(t)} - x_i^{(t)}) + \alpha\epsilon_i, \tag{4}$$

where the first term is the firefly's current position, the second term is the attraction term. And the third term is the random component. The Cartesian distance between two fireflies i and j is represented in equation four as r_{ij}. The β_0 term means the attractiveness when there is no distance between the two fireflies. This term is usually set to 1, and if it's set to 0, FA becomes a random walk. The γ term represents a variation in attractiveness as a light absorption coefficient, usually with a value varying from 0.1 to 10, which is essential in adjusting the convergence speed. The third term is made up

of α, a randomization parameter in $[0, 1]$ and ϵ_i a uniformly distributed random vector with entries in $[0, 1]$, the same dimensionality as the position.

IV. A PSO Method

In 2018 a paper was written by Lv et al., called *Test cases generation for multiple paths based on PSO algorithm with metamorphic relations* to improve the efficiency of test case generation of already proposed PSO methods [1]. This paper combines PSO with Metamorphic Relations (MRs) to ease the necessity of an Oracle question. An MR represents the relationship between inputs and outputs after program execution. The initial test suite is generated by the PSO algorithm, which MRs utilize to develop new test cases. Since the PSO algorithm is not solely generating test cases, the MRs are able to create new test cases more efficiently while using fewer resources.

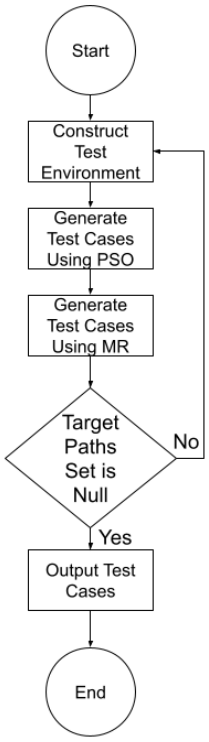

Fig. 3. Architecture for test case generation based on PSO & MRs [1].

The proposed method from Lv et al. is made up of three main parts: (1) constructing the test environment; (2) generating test cases based on PSO; (3) generating test cases based on MRs. The overarching architecture of their proposed method is shown in Fig. 3:

(1) Constructing the test environment: This step lays the foundation for the proposed method. A fitness function is constructed in this block, a static analysis is done on the program under test (PUT), produces the MRs, and instruments PUT.

(2) Generating test cases based on PSO: This step is vital for the proposed method. Since the first test case and some cannot be generated by MRs alone, PSO must pick up the slack. Each particle in PSO represents one test case. The particle evolves towards a favourable solution after the population is initialized in the search space mapped from the input domain of test cases. The fitness of the position of each particle is calculated by instrumenting the PUT onto the test case associated with our particle.

(3) Generating test cases based on MRs: The PSO-based generated test case acts as an origin test case (OTC) to be further used to create a follow-up test case (FTC). The FTC is stored if it traverses the targeted path. On the other hand, if other test cases have not travelled the FTC's executed path, then we use the FTC as an OTC to create a new FTC. This process will repeat until there are no new paths to traverse for the FTCs, and they will stop being generated by MRs.

The experiments and results from this paper present an interesting new way to generate test cases for numerous programs automatically. It was found that sometimes test cases could be caused solely by MRs and didn't require PSO to explore the search space cutting down on computational investment, thus successfully improving the efficiency of generating test cases. However, they do have a few pitfalls. First, MRs need to be constructed before their method is used, which requires real testers with domain knowledge about the PUT, which drastically affects its applicability to a wide range of cases compared to other proposed methods. Second, it was found that performance could vary significantly depending on the tested path order and the used MRs. Lv et al. believe this topic requires further study due to the found promise in their proposed method and the effectiveness it has brought to automatic test case generation, even given its niche requirements.

V. GA Method

Automatic Test Case Generation based on Genetic Algorithm and Mutation Analysis was written by Haga and Suehiro in 2012, and it proposed a way of generating software test cases (focusing on unit testing) by combining genetic algorithms and mutation analysis [3]. Initially, test cases are generated randomly, and over time the test cases become polished by a GA. The sufficiency of the test suite is measured by mutation scores, which in software testing come from mutation analysis (MA). A MA adds intentional errors into the program, and the mutation score is determined by how many of these errors the test suite was able to detect. Redundant test cases can be removed from the generated ones by a detection matrix that shows which test cases can detect specific mutations. If multiple test cases capture the same modifications, then the excess test cases can be removed. If the mutation score does not meet particular criteria, test cases need to be added or modified to progress the score. The writers of the paper claimed their proposed method obtained 100% boundary and branch coverage.

Haga and Suehiro's method architecture can be seen in Fig. 4. The steps for executing their proposed method are as follows: (1) Within the predefined range of the input data randomly generate initial test cases; (2) Apply mutant

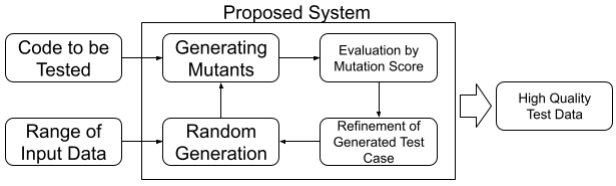

Fig. 4. Architecture for test case generation based on GA & MA [3].

operators to generate mutants and from these generated mutants construct a corresponding detection matrix; (3) Without affecting detection ability use the generated detection matrix to reduce the number of test cases; (4) Calculate the mutation score. Exit the procedure if there is convergence; (5) If we've not exited the system, continue using GA to generate new test cases and repeat the whole process.

The paper presented used GA and MA methods for automatically generating valuable test cases. They showed that their proposed method with GA took 20% longer, but the quality of the generated test cases was much higher. The main takeaway is that the proposed method can generate superior quality test cases compared to hand-crafted ones automatically. However, issues remain with the proposed system: (1) only usable with numerical data; the generated test cases cannot achieve (2) complex coverage measures; (3) a limited number of mutant operators were tested, leaving room for further investigation in the future.

VI. ACO Method

In 2005, a paper written by Li and Lam called, *Software Test Data Generation Using Ant Colony Optimization* introduced an Ant Colony Optimization (ACO) algorithm paired with UML Statechart diagrams for generating test data to be used for testing program execution [4]. The proposed method starts by converting a UML Statechart diagram representing the system we're testing into a directed graph. The ants explore the created graph to generate test data to meet a test coverage adequacy criterion. Overall, the presented algorithm achieved solutions to all states' problems, and each path was feasible. However, they could not generate a test suite containing non-redundant test cases, i.e. have the smallest possible number of them.

The method by which they convert a UML Statechart diagram to a directed graph is not given in the paper. Still, they provide information on how the ants explore the generated directed graph. Take $G = (V, E)$ to represent our directed graph where V is the number of vertices, and E is the number of edges. At the beginning of exploration, a group of ants are dispatched simultaneously to explore. Each ant k at a vertex α on the graph is linked to their own four tuple (S_k, D_k, T_k, P) where S_k is the vertex track set, D_k is the target set, T_k is the connection set, and P is the pheromone trace set. The pheromone trace set P is a public set all ants share while the sets S_k, D_k, and T_k are specific to each ant. The vertex track set acts as a walking history for the ant. The target set indicates all always connected vertices to the currently explored vertex α. The connection set represented the neighbouring vertices connected to our current vertex α. It also keeps track of all spanning edges from α. The pheromone trace set indicates to the ant the pheromone levels at all neighbouring vertices. Ants explore the search space by these properties and terminate when the union of all S_k sets for each deployed ant contain all of the graph's vertices. This indicates that all states have been visited. If the algorithm does not terminate based on this criteria, we terminate based on a search upper bound, indicating we failed to find a full-coverage solution.

The following steps can summarize the exploration of an ant k: (1) Update the track by adding the current vertex α onto the stack set S_k; (2) Determine T_k by evaluating all connection to the current vertex α; (3) For possible transitions to other vertices in the graph, since the trace, i.e., acquire the pheromone levels at the end of these connections; (4) Select a destination by the lowest pheromone level or randomly if they're the same; (5) Update the pheromone level at the current vertex; (6) Move to the found destination vertex; (7) Repeat the whole process until one of the termination criteria are met.

VII. CS & BCO Method

Lakshminarayana and SureshKumar wrote a paper in 2019 titled, *Automatic Generation and Optimization of Test case using Hybrid Cuckoo Search and Bee Colony Algorithm* proposing a hybrid CS and BCA (CSBCA) method for generating and optimizing test cases with minimal execution time focusing on path coverage from combinational UML diagrams [2]. The writers compared their proposed method with PSO, CS, BCA, and FA. The comparison study between the proposed method and the comparison methods showed CSBCA to outperform all comparison methods. However, the writers mention that this work was only tested on the bank ATM problem and should be applied to more complex issues in the future to ensure its validity.

In the proposed method, they first convert a statechart diagram to a statechart diagram graph and convert a sequence diagram to a sequence graph, which creates a fitness landscape for the meta-heuristic evolutionary methods to explore. The conversions presented are not discussed in detail. The population and number of iterations are fixed based on the number of test cases. The fitness function appears to be problem-dependent for the proposed method, and in their case, they're testing it on the bank ATM problem. The procedure of the algorithm is as follows: (1) randomly initialize a population of solutions; (2) evaluate the fitness of the generated solutions; (3) Rank the generated solutions based on their fitness; (4) remove the less fit half of the population; (5) perform BCA and CS in parallel with the remaining population; (6) if the generated solutions from either algorithm are better than the overall best found so far, then we replace it; otherwise, we discard the newly generated solution; (7) repeat processes 3-6 until the termination criterion is met; (8) upon termination the best solution in the population will be our overall solution. The CSBCA's proposed architecture can be seen in Fig. 5.

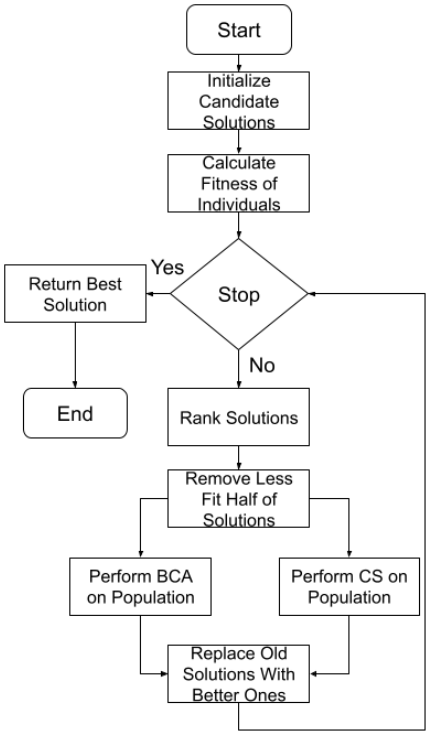

Fig. 5. Procedure for test case generation/optimization based on CSBCA [2].

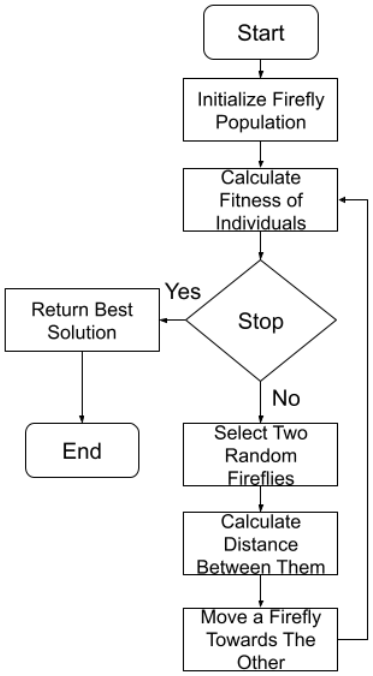

Fig. 6. FA architecture used in automatic test case generation [16].

VIII. FA Method

Sahoo et al. wrote a paper in 2016 called, *A Firefly Algorithm Based Approach for Automated Generation and Optimization of Test Cases* to effectively and efficiently automatically generate test cases and test data using FA [16]. The writers apply their proposed method, the Bank ATM withdrawal method, to perform successfully on the objective task. They claim that their proposed method outperformed PSO, bat, harmony search, and cuckoo search for both runtime and accuracy. In the future, they hope to test their proposed method on more extensive problems and improve it in such a way as to increase code coverage. The proposed method follows the FA's procedure almost entirely except for the fitness function, which defines the light intensity for each firefly. The fitness function is directly related to their test problem. The best firefly at the end of the evolution acts as our new test data, and for each iteration, we get more and more test cases generated over time. The Firefly Algorithm presented and used in this paper is illustrated in Fig. 6.

IX. Conclusions & Future Work

There are numerous Evolutionary Computation (EC) strategies for improving software testing in the field of Software Engineering (SE). Many of these strategies help reduce the amount of human intervention in testing and save a great deal in development costs. These strategies allow us to find reasonable possible solutions to testing software. Since it is almost impossible to test every piece of software, EC gives us a way of trying the most important aspects.

Out of the various presented EC methods for automatically generating test cases, PSO and GA are the most used and well-researched strategies for this particular topic. Since PSO and GA are the most researched EC methods and the most studied for this topic, they are a prime pick for this problem domain. However, if you'd like you to write in areas not explored, then sticking with one of the other methods or combining a couple may be your best shot. The different explored strategies in this paper are more niche and not well researched, leaving lots of room for improvement and scientific exploration. The future of applying EC to these types of problems seems to come down to combining an EC algorithm with another algorithm from a different sub-field of computer science to help with the process. The difficulty with applying EC to any problem revolves around generating an appropriate numerical fitness function.

In conclusion, EC strategies and their application to software testing show promise and have been proposed in many areas of software testing. The methods proposed using EC may not be the most efficient way to generate test cases within the scientific community, but we can get a definitive answer with more research. In addition, as hardware improves in the future, EC will become more efficient. Future work involves finding ways to represent software testing problems as search-based problems for Evolutionary Computation (EC) strategies.

REFERENCES

[1] X.-W. Lv, S. Huang, Z.-W. Hui, and H.-J. Ji, "Test cases generation for multiple paths based on PSO algorithm with metamorphic relations," *IET Software*, vol. 12, no. 4, pp. 306–317, Aug. 2018. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1049/iet-sen.2017.0260

[2] P. Lakshminarayana and T. V. SureshKumar, "Automatic Generation and Optimization of Test case using Hybrid Cuckoo Search and Bee Colony Algorithm," *Journal of Intelligent Systems*, vol. 30, no. 1, pp. 59–72, Jul. 2020. [Online]. Available: https://www.degruyter.com/document/doi/10.1515/jisys-2019-0051/html

[3] H. Haga and A. Suehiro, "Automatic test case generation based on genetic algorithm and mutation analysis," in *2012 IEEE International Conference on Control System, Computing and Engineering*. Penang, Malaysia: IEEE, Nov. 2012, pp. 119–123. [Online]. Available: http://ieeexplore.ieee.org/document/6487127/

[4] H. Li and C. P. Lam, "Software Test Data Generation using Ant Colony Optimization," vol. 1, p. 5, 2005.

[5] R. K. Sahoo, D. Ojha, D. P. Mohapatra, and M. R. Patra, "Automated Test Case Generation and Optimization : A Comparative Review," *International Journal of Computer Science and Information Technology*, vol. 8, no. 5, pp. 19–32, Oct. 2016. [Online]. Available: http://aircconline.com/ijcsit/V8N5/8516ijcsit02.pdf

[6] M. Harman, "Software Engineering Meets Evolutionary Computation," *Computer*, vol. 44, no. 10, pp. 31–39, Oct. 2011. [Online]. Available: http://ieeexplore.ieee.org/document/6036090/

[7] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, vol. 4. Perth, WA, Australia: IEEE, 1995, pp. 1942–1948. [Online]. Available: http://ieeexplore.ieee.org/document/488968/

[8] S. Helwig and R. Wanka, "Particle Swarm Optimization in High-Dimensional Bounded Search Spaces," in *2007 IEEE Swarm Intelligence Symposium*, Apr. 2007, pp. 198–205.

[9] Y. Shi and R. C. Eberhart, "Parameter Selection in Particle Swarm Optimization," in *Evolutionary Programming VII*, V. W. Porto, N. Saravanan, D. Waagen, A. E. Eiben, G. Goos, J. Hartmanis, and J. van Leeuwen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, vol. 1447, pp. 591–600. [Online]. Available: http://link.springer.com/10.1007/BFb0040810

[10] Y. Shi and R. Eberhart, "A modified particle swarm optimizer," in *1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98TH8360)*, May 1998, pp. 69–73.

[11] M. Melanie, "An Introduction to Genetic Algorithms," p. 162, 1996.

[12] M. Dorigo, V. Maniezzo, and A. Colorni, "Positive feedback as a search strategy," p. 23, Jun. 1991.

[13] X.-S. Yang and S. Deb, "Cuckoo Search via Levy Flights," *arXiv:1003.1594 [math]*, Mar. 2010, arXiv: 1003.1594 version: 1. [Online]. Available: http://arxiv.org/abs/1003.1594

[14] D. Teodorović, "Bee Colony Optimization (BCO)," in *Innovations in Swarm Intelligence*, J. Kacprzyk, C. P. Lim, L. C. Jain, and S. Dehuri, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, vol. 248, pp. 39–60, series Title: Studies in Computational Intelligence.

[15] X. Yang, *Nature-inspired metaheuristic algorithms*, 2nd ed. Luniver Press, 2010.

[16] R. Sahoo, D. Mohapatra, and M. Patra, "A Firefly Algorithm Based Approach for Automated Generation and Optimization of Test Cases," *INTERNATIONAL JOURNAL OF COMPUTER SCIENCES AND ENGINEERING*, vol. 4, pp. 54–58, Aug. 2016.