Automatic Fault Localization using Artificial Bee Colony Algorithm

Implementation of the paper (Paper by Linzhi Huang and Jun Ai

I. Introduction

With the coming of this technological age caused by the second Industrial revolution or the IT revolution, everything is being automated, everything is being mechanised. We build machines and make softwares which can replicate any work that a human does and can replace human intervention or human presence. Oftentimes these softwares, and machines which also have some sort of software in them, have bugs in them. These bugs need to be identified and removed in order for the software to work as required. This process of debugging usually takes a long time. This process is also an extremely expensive one. Many people have tried to automate this process of localization of these bugs (faults). A remarkable approach is the one proposed by Linzhi Huang and Jun Ai in their paper "Automatic Software fault localization based on Artificial Bee Colony". This project is an attempt at implementing the scheme/ ideas presented in this paper.

[2] The artificial bee colony (ABC) algorithm was originally proposed by Dervis Karaboga in 2007 under the inspiration of the collective behavior of honey bees. This evolutionary/genetic algorithm is used for numerical optimization and many other mathematical problems. This is an especially good algorithm because it has a lower chance of getting stuck in a local optima.

Honey bees have an interesting way to find food. Some honey bees look for nectar in flowers nearby. When some of these bees find some flowers with nectar, they come back to their nest and do this thing called the *waggle dance*. This dance, as studied by many scientists, has the information about the whereabouts and the quality of the nectar. Following this, other bees then go around that particular nectar source to look for an even better one. This process continues and finally yields to an optimal food source for the bees. All this while, some other bees go around wondering, looking for a food source without any prior knowledge. Thus they may get a source of food that is better than any previously found. This behaviour reduces the chances of the bees getting stuck near a food source that may be sub-optimal. Thus reducing the chance of getting stuck to a local optima.

II. Explanation of the Solution

There are food sources, employed bees, unemployed bees and the nectar amount in the food source in the *honey -bee-optimal-food-source-search-scenario*. The employed bees(N_e) are the ones which are out there looking for food sources. The unemployed bees (N_u) are the ones sitting in the nest, waiting for the currently employed bees to return so that they can go out further beyond the current search to explore better food sources, if necessary.

To establish an analogy with the search scenario, in our system, each program unit is a food source, and the nectar amount corresponds to the fitness value of that particular program unit. Each test case is a bee. The test cases which are currently running correspond to employed bees.

The test cases that fail or pass, have an execution trace. This execution trace basically has the information as to which all program units were executed when this test case passed or failed. We do this - many times, over many test cases- to see a pattern and thus to localize the fault. There are basically 6 steps:

1. The set of initial test cases have to be executed. After this the test case cost value is calculated and the first N_e test cases are selected.
2. New employed bees search for new solutions by mutating the initial test case set. We use the following for mutation:
$$V_{i,j} = X_{i,j} + \varphi_{i,j} (X_{i,j} - X_{k,j})$$
where $V_{i,j}$ is the new test case, $X_{i,j}$ is the current test case, both $X_{i,j}$, $X_{k,j}$ belong to initial test cases, $\varphi_{i,j}$ is a random value between -1 and 1, k and j are randomly chosen numbers from {1,2,...D}, {1,2...N_e} respectively where D is the Dimension of the test case, Ne is number of initial test cases; and V belongs to S(Search Space).
3. The better test case amongst the newly generated and the older one (i.e. amongst $V_{i,j}$ and $X_{i,j}$) is chosen.
4. The unemployed bees (represented by the newly generated test cases) now choose the individual bee (the employed test case) that maintains a relatively low cost as the leader bee. After this the unemployed bees search for a solution near the leader bee's solution space using a method similar to step 2. The fitness value of these solutions is then calculated and the selection operator is used to select amongst all these test cases.
5. The best fitness value is updated after each iteration.
6. If the search time is larger than a particular threshold, the employed bee becomes an unemployed bee.

Fitness Function:

To calculate the fitness value we need a fitness function. There are two factors which are considered while calculating the fitness value.

The first factor is the correlation coefficient:

$$S_T(k) = \frac{\frac{a_{11}(s_k)}{a_{11}(s_k)+a_{01}(s_k)}}{\frac{a_{11}(s_k)}{a_{11}(s_k)+a_{01}(s_k)} + \frac{a_{10}(s_k)}{a_{10}(s_k)+a_{00}(s_k)}}.$$

$a_{11}(s_k)$ is the number of test cases that execute the basic unit s_k and fail, $a_{10}(s_k)$ is the number of test cases that execute the

basic unit s_k and pass, $a_{01}(s_k)$ is the number of test cases that do not execute the basic unit s_k and fail. $a_{00}(s_k)$ is the number of test cases that do not execute the basic unit s_k and pass.

The second factor is the program dependence relationship. If, out of 2 program units - 'a' and 'b', if 'a' has some sort of control or data dependence on 'b' then it contributes some score to the score of 'b'.

Together these factors give us a fitness function which we use in the ABC Algorithm. We make sure to make quite some hyper parameters because we may have to later tune these to get better results.

$$Score(k) = S_T(k)\ \pi_{i=1\ to\ N}\ a_{ik}$$

III. IMPLEMENTATION

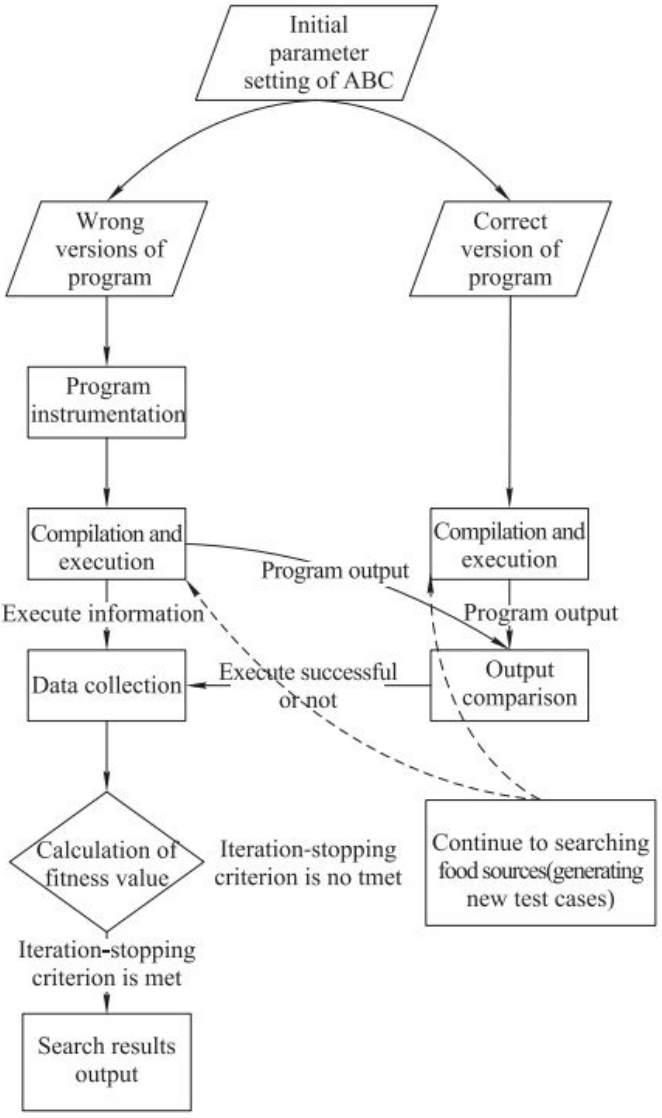
The following steps are then taken, the flow of which is shown in flowchart 1.1 on the RHS, which is taken directly from the original paper:

1. The wrong version of the program is first instrumented. What this means is that we insert probes which simply monitor if the execution reaches a particular line.
2. Then the correct and the wrong versions of the program are run on the same test case.
3. Then their output is compared and the data about which lines were executed is collected.
4. Then the fitness values of this particular test case is calculated and the whole procedure is done again, abiding to the rules of the ABC algorithm, i.e. new test cases are generated and all the steps mentioned in the previous section of the report are taken.
5. This goes on until the stopping criterion is met.

Fault: statements 4 should be: if ($x < y$); void mid (){		Test case						
		1,1,1	1,2,3	1,3,2	2,1,3	2,3,1	3,1,2	3,2,1
int x, y, z, m;								
read (x, y, z);	s1	1	1	1	1	1	1	1
$m = z$;	s2	1	1	1	1	1	1	1
if($y < z$)	s3	1	1	1	1	1	1	1
if($x > y$)	s4	0	1	0	1	0	1	0
$m = y$;	s5	0	0	0	1	0	1	0
else if($x < z$)	s6	0	1	0	0	0	0	0
$m = x$;	s7	0	1	0	0	0	0	0
else								
if($x > y$)	s8	1	0	1	0	1	0	1
$m = y$;	s9	0	0	0	0	0	0	1
else if($x > z$)	s10	1	0	1	0	1	0	0
$m = x$;	s11	0	0	0	0	1	0	0
print(m);}	s12	1	1	1	1	1	1	1
status		P	F	P	F	P	F	P

Table 1.1

The table 1.1 is taken from the original paper and it provides traces of execution of test cases in the form of a truth table. We see, for example, for the first test case, which passed, the line numbers 1, 2 and 3 were executed and the rest were not. This information is used by our algorithm to successfully detect patterns and to further localize faults by calculation of fitness value of the iteration of test cases.



Flowchart 1.1

The following are the details of the project. We use the TCAS data set that the authors of the original paper propose. The TCAS dataset has 41 wrong versions, 1 correct version with 1608 test cases. We have chosen some versions of programs as described in the paper and instrumented them and placed in folder instrumented for our experiment.

The current instrumented code is the sixth version of the wrong 'c' program right now. Generate the TCAS program executables using following commands before running algorithm

1. "Run gcc gcc tcas.c -o tcas_right.out" to get the executable of correct version
2. "Run gcc gcc tcas_1.c(change filename as needed) -o tcas_wrong.out" to get the executable of correct version

Run the main python file (tcas.py) which has algorithm to run

To change the hyper parameters of execution give the following parameters(or give the --help with the python file to get more details):

1. -S, --InputVectorSize: Input Vector Size.
2. -I, --Iterations: Number Of Iterations.

3. -Ne, --NoEmployedBeesPI: Number of Employed Bees per iteration.
4. -F, --FitnessChangeLimit: Abandon test case if change in fitness for many iterations is less than this.")
5. -D, --DependencyFitness: Dependency Fitness multiple value.
6. -A, --AbandonAfter: Abandon after trying for this many iterations.
7. -DE, --DivisionError: Add this to the denominator for removing division by zero error.
8. -P, --Probes: Number of Probes.

IV. RESULTS

We see that from the above method we get a score value for different program units. After several iterations, we see that the suspicion score of a particular program unit or a few particular program units is higher. This is the result of our fault localization problem.

We tried to replicate and further improve the results of the original authors of the paper, however we could not do so to the fullest of our expectations because the original paper does not share the exact details of their initial implementation. We had to assume many factors and have a lot of hyper parameters. We were able to instrument the initial program from the TCAS dataset and successfully make the program for the complete system. The results that we got from our implementation were suboptimal.

Wrong version	Fault type	Fault position	Location result	Iteration number
4	Logic mutation	Line 75	Probe 9 Lines 73 – 76	10
6	Operator mutation	Line 104	Probe 11 Lines 102 – 105	5
12	Logic mutation	Line118	Probe 0 Lines 114 – 118	29
20	Operator mutation	Line 72	Probe 8 Lines 66 – 72	15
22	Operand mutation	Line 72	Probe 8 Lines 66 – 72	70
26	Code missing	Line 118	Probe 0 Lines 114 – 118	70
27	Code missing	Line 118	Probe 0 Lines 114 – 118	37
29	Operand mutation	Line 63	Probe 13 Lines 61 – 64	92

Table 1.2

The table 1.2 is the results table from the original paper. We tested our program for the first few wrong versions of the programs only. We got different results from the original paper. The results were however quite close. The improvement of the method, falls under the section of future work for now.

V. LESSONS LEARNED

The initial challenge while selecting this project was knowing the vastness of it and having the courage to knowingly experiment with something with several implementational holes.

Rolling along the course of this project I learned a whole new set of algorithms - the genetic algorithms. It uses a non conventional approach to computer science. It uses genetic biology and applies it to mathematical problems. It observes what nature is doing and tries to imitate that on different problems. How successful these methods have been in the past and are being in the present is quite amazing.

I got to experience the power of artificial intelligence first hand. Localizing a logical fault is something that can only be comprehended by the human brain. Making a model to detect such a fault shows the potential that artificial intelligence holds.

I learned to write and manage long codes. I got to interface 2 programming languages, which I had previously not done.

Apart from technical abilities, I learned to work in a team. This was the first group project. I never understood how people work in a group on something that has interdependencies. I not only understood it, I got to experience it first hand.

VI. TEAM MEMBERS

The following are the names of people who made this project possible and their initial contribution decision:

1. Aabhaas Gupta
   a. To analyse and design ABC algorithm associated with the localization problem.
   b. To define and select the fitness function of ABC algorithm related to the localization problem.
2. Sai Harshith Rao Jupally
   a. To collect the execution data (TCAS programs in Siemens Suite).
   b. To convert this data into a format for the ABC algorithm to run.
3. Harsh Lalwani
   a. To setup the environment for the system and the experiment.
   b. To design the experimental setup.
   c. To work on result analysis.

VII. PERSONAL CONTRIBUTIONS

I had to analyse the Artificial Bee Colony Algorithm and make it such that it fits the scenario of our fault localization problem. I helped the group understand the problem statement and thus come to clever ways of solving challenging tasks.

VIII. REFERENCES

[1] L. Huang and J. Ai. “Automatic software fault localization based on artificial bee colony”. In: Journal of Systems Engineering and Electronics 26.6 (Dec. 2015), pp. 1325–1332. ISSN: 1004-4132. DOI: 10 . 1109 / JSEE.2015.00145.

[2] D. Karaboga, B. Basturk. A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm. Journal of Global Optimization, 39(3), 459 – 471.