2019 IEEE International Conference on Systems, Man and Cybernetics (SMC)
Bari, Italy. October 6-9, 2019

A Test Case Prioritization Approach Based on Software Component Metrics

Dennis Sávio Silva[1], Ricardo Rabelo[1], Pedro Santos Neto[1], Ricardo Britto[2] and Pedro Almir Oliveira[3]

Abstract— **The most common way of performing regression testing is by executing all test cases associated with a software system. However, this approach is not scalable since time and cost to execute the test cases increase together with the system's size. A way to address this consists of prioritizing the existing test cases, aiming to maximize a test suite's fault detection rate. To address the limitations of existing approaches, in this paper we propose a new approach to maximize the rate of fault detection of test suites. Our proposal has three steps: i) infer code components' criticality values using a fuzzy inference system; ii) calculate test cases' criticality; iii) prioritize the test cases using ant colony optimization. The test cases are prioritized considering criticality, execution time and history of faults, and the resulting test suites are evaluated according to their fault detection rate. The evaluation was performed in eight programs, and the results show that the fault detection rate of the solutions was higher than in the non-ordered test suites and ones obtained using a greedy approach, reaching the optimal value when possible to verify. A sanity check was performed, comparing the obtained results to the results of a random search. The approach performed better at significant levels of statistic and practical difference, evidencing its true applicability to the prioritization of test cases.**

I. INTRODUCTION

Regression testing is a type of software testing that checks if changes performed in a system did not insert faults or led to unexpected effects [1]. The most common approach for regression testing is the complete retest of the system. However, as a software expands, the number of test cases tends to increase, discouraging this practice. Some approaches used to solve this problem are the test case selection and the test case prioritization [2].

In Silva et al. [3], we proposed an approach based on the prioritization and selection techniques for improving regression testing. The approach obtains a subset of test cases from a software system's original test suite. To do so, it complies with an imposed time constraint related to the time to execute the test cases and maximizes the total criticality of the selected test cases. Although the approach has shown reasonable results, it has limitations; the use of a selection technique can decrease the potential of fault detection of a test suite. In Silva et al. [4] we proposed a new approach aiming to address the limitations of our previous approach. To do so, we employed test case prioritization to improve the fault detection rate of test suites. The approach uses fault-proneness-related software metrics and test coverage information to obtain the criticality of the test cases. It solves the test case prioritization problem using the following formulation: Given a set of test cases, to find an ordering that maximizes the fault detection rate of the regression testing, considering information available during the test execution, like the fault detection history, as well as metrics obtained through code analysis and related to the fault-proneness of the code components.

This paper extends our previous works. The main aim is to perform a practical study on an approach for test case prioritization that contemplates metrics of the software components and information about the coverage and cost of the test cases. To achieve this, we inserted in the approach a metric that considers the cost of the test cases together with the rate of fault detection of the test suites, allowing to better measure the benefits obtained with the approach. We tuned the prioritization algorithm's parameters, allowing to obtain better results without an excessive increase of computational cost for the approach. In the experimentation, we performed a sanity test, through the evaluation of the statistical and practical difference from the results of the approach to the results from a random search.

The approach described in this paper has three steps and uses the information available during the execution of the regression testing to prioritize the test cases, aiming to improve the fault detection rate of the test set. In the first step, a fuzzy inference system [5] assigns a criticality value to each code component, reflecting its fault-proneness. In the second step, the criticality (fault detection potential) of the test cases is obtained on the basis of the criticality of the software components they cover. In the third step, an Ant Colony Optimization (ACO) [7] algorithm is used to prioritize the test cases considering their criticality, execution time and fault detection history. An evaluation of the approach was performed, using programs from a public repository. Among the main contributions of this paper, we can cite the use of the APFD$_C$ - a metric derived from the APFD, which considers the cost of the tests together with the rate of fault detection - in the prioritization approach; a series of experiments, aiming to tune the parameters of the algorithm used in the prioritization of the test cases; a comparison - in terms of statistic and practical significance - of the results obtained with the proposed approach and the results obtained with a random approach; and a comparison of the results of the approach to the original (unchanged) execution order of the test cases, to an ordering obtained with a greedy algorithm,

[1]Federal University of Piauí, Computer Science Department, Teresina, Piauí, Brazil dennissavio@gmail.com, {ricardoalr, pasn}@ufpi.edu.br
[2] Blekinge Institute of Technology, Karlskrona, Sweden ricardo.britto@bth.se
[3] Federal Institute of Maranhao, Pedreiras, Maranhao, Brazil pedro.oliveira@ifma.edu.br

and to the optimal order of the test suite (when obtainable by an exhaustive search), employing a set of real open programs often used in works about regression testing.

The remainder of this paper is organized as follows: In Section II the theoretical background is presented. Section III presents related works. Section IV introduces the proposed approach. Section V describes the preliminary studies to tune the parameters of the approach and presents the results of the experimentation. Section VI presents threats to validity. Section VII presents the conclusions and directions for future works.

II. THEORETICAL BACKGROUND

A. Regression Testing

The Regression testing checks if modifications performed in a new software version do not affect the behavior of the already existent and unchanged software components in an unexpected way [2]. The simplest approach to regression testing is the execution of all the test cases of the system. However, when the program grows, the number of test cases tends to increase, and this approach can consume excessive time and resources, or even become unfeasible. Many techniques were already proposed aiming to improve the cost-effectiveness of the regression testing. Some of these techniques can be classified in test case selection or test case prioritization. Test case selection creates a subset of the program test cases considering some criteria. The unselected test cases are not definitely removed from the system and can be reused in future tests [2]. Test prioritization does not involve the non-execution of test cases, but the search for an execution order for the test cases that maximize properties like the fault detection rate or the rate of fault coverage, allowing the test objectives to be reached earlier.

B. Ant Colony Optimization

Ant Colony Optimization is an optimization model inspired by the foraging behavior of real ants [7]. More specifically, in the capacity of the ants of finding the shortest path from the nest to a food source using pheromone. Pheromone is a chemical substance distributed by real ants when walking in search of food, and it guides the ants whose will follow the path later. Paths with higher pheromone concentration have more chance of being chosen. This behavior inspires the optimization algorithm, where the ants are computational agents whose paths represent solutions of a problem. The ACO algorithm involves two basic procedures: Building of the solution (m ants build in parallel m solutions to the problem) and Pheromone update (based on an evaluation function that measures the quality of the solutions, and better solutions receive a higher pheromone deposit). In addition to the pheromone information, a heuristic function is used to improve the building of solutions to the problem.

C. Fuzzy Inference Systems

Fuzzy logic is suitable for situations where the information is uncertain or imprecise and words are better to represent concepts than numbers [5]. While the traditional logic works with absolute truth or falseness, the fuzzy logic deals with partial levels of truth. Fuzzy inference systems deal with qualitative information, and their inputs and outputs are understandable from a linguistic viewpoint. Fuzzy systems are based on linguistic production rules in the *If... Then...* format. Fuzzy set theory and fuzzy logic provide the mathematical basis to deal with imprecise information. Fuzzy sets are a generalization of classical sets, whose elements can have partial membership degrees (μ).

Fuzzy inference systems are composed of: **Input interface (fuzzification)**: receives the input variables of the system, and converts the crisp input values into fuzzy sets, represented by linguistic terms, such as ”high”, ”low”, etc.; **Fuzzy inference system**: performs the reasoning about the rules and facts provided to the input interface, to obtain the outputs; **Knowledge base**: Composed by a rule base (set of fuzzy rules to map the relations between input and output values) and a database (stores the membership functions and the universe of the variables); and **Output interface (defuzzification)**: the result obtained in a fuzzy inference system is a new fuzzy set. In order to this result be understood by the specialist, it is necessary that it passes by a process of extracting the most representative crisp result of the output fuzzy set. The proposed approach applies a Mamdani fuzzy inference System model [8], which allows defining a rule base linguistically, without using numerical data. The production rules in Mamdani models have fuzzy sets in their antecedents and consequents.

III. RELATED WORK

Mukherjee and Patnaik [6] performed a survey to investigate test prioritization. It was found that to evaluate the solutions, the most applied metric was the APFD (Average Percentage of Faults Detected). The mapping identified a significant use of coverage-based methods, like the one proposed by Elbaum, Malishevsky, and Rothermel [10]. But only the coverage cannot assure an improvement in the fault detection rate, so it is recommended to use more than one criteria in the prioritization [9]. The fault detection history is widely used in regression testing optimization, as evidence of the test capability of fault detection. An example of work applying the fault history is [11].

The execution time is also often used. Gao et. al. [12] prioritize test cases through an ACO algorithm that considers it together with the fault history and severity. Bharti and Singhal [13] implemented an ACO based test case selection and prioritization technique to deal with time restrictions in the regression testing.

The criticality of the test cases is also a prioritization criterion, as seen in Acharya, Khandai and Mohapatra [14] which uses the BCV (Business Criticality Value) metric to represent the contribution of a software’s functions to the business.

In works which employ the coverage metric, the aim is to maximize the rate at the which the code is covered by the test cases. There are metrics directed to measure this rate, such as the APBC (Average Percentage Block Coverage), the APDC (Average Percentage Decision Coverage) and the APSC (Average Percentage Statement Coverage) [15]. In the proposed approach, the generated test suites are evaluated by their fault detection rate (through the APFD metric, the Average Percentage of Fault Detection), which is not directly related to the coverage. So, the coverage is considered in the calculation of the test case criticalities, trying to benefit test cases which cover larger parts of the code.

Furthermore, the execution time (cost of the test case) and the fault detection history (privileging test cases which are proven effective in regression testing) are employed in the proposed approach. The attribution of criticality levels to the test cases based on information from the own project also was not found in previous works about test case prioritization. Previous researches about test prioritization did not put together all these inputs.

The use of the history of fault detection, execution time, coverage, cost and criticality to prioritize test cases is very common in regression test prioritization works. The optimization of the rate of fault detection is also very performed. But it is not of our knowledge any previous work that has linked all these information and criteria. Our approach put together many aspects consolidated in works about regression testing to create test suites based on their potential rates of fault detection, using a metaheuristic in order to improve the obtained results and optimize the regression testing activity.

IV. PROPOSED APPROACH

The proposed approach is based on test case prioritization (Figure 1). It starts inferring the software components criticality, which is used to obtain the criticality of the test cases, that by their turn are employed in the test prioritization.

Step I - Calculating the criticality of the software components: The criticality of the software components is inferred using a fuzzy inference system, considering software metrics related to the fault-proneness, and has as aim to identify parts of the software under test which should be verified as soon as possible, allowing to prioritize the execution of tests which cover more critical sections of the software. The use of a fuzzy system simplifies the model to be optimized by the metaheuristic, reducing the search space and easing the interpretation of the results. The inputs of this fuzzy system are the software metrics well known in software engineering: **Cyclomatic complexity** (**CC**) [16], the number of possible execution paths of the code; and **Instability** (**I**) [17], the relationship level among the software components, determined by the expression $I = C_e/(C_e + C_a)$, where C_e is the efferent coupling (number of components to which the evaluated component references); and C_a is the afferent coupling (number of components that reference the evaluated component).

The input values were normalized to the 0-10 interval. The output variable of the proposed fuzzy inference system is the criticality of the software components. The rule base (Table I), the inputs (Figures 2a and 2b) and output (Figure 2c) variables of the fuzzy inference system were defined considering our knowledge until now and validated in empiric experiments with software engineering specialists.

TABLE I: Rule base of the fuzzy inference system

		Instability				
		Very Low	**Low**	**Medium**	**High**	**Very High**
CC	**Very Low**	Very Low	Very Low	Low	Low	Medium
	Low	Very Low	Low	Low	Medium	Medium
	Medium	Low	Low	Medium	High	High
	High	Low	Medium	High	High	Very High
	Very High	Medium	Medium	High	Very High	Very High

Step II - Calculating the criticality of the test cases: The criticality TC of a test case j is calculated using as inputs the criticality of the software components and the coverage information of j, as represented by the formula:

$$TC_j = \frac{\sum_{i=1}^{nj} FC_i * x_{i,j}}{\sum_{i=1}^{nj} FC_i} \tag{1}$$

where: i is the index of a component covered by the test case j; nj is the number of components covered by the test case j; FC_i is the criticality of the component i; $x_{i,j}$ is the coverage percentage of the component i by the test case j (the ratio between the component lines executed by the test case and the total of lines in the component).

Step III - Test case prioritization: The inputs of this step are the test cases criticality, the fault detection history and the execution time of the test cases. The prioritization process was modeled as the traveling salesman problem, which consists of finding the shortest closed path in a graph, visiting every node (test cases in our case) exactly one time. To solve this problem, it was used the *MAX-MIN Ant System* (MMAS) [18], an ant colony optimization algorithm.

The algorithm evaluates the solutions using two metrics: APFD and APFD$_C$ [10]. APFD (Average Percentage of Faults Detected) represents the rate at which the faults are detected by a test suite, and is defined by the formula:

$$APFD = 1 - \frac{TF_1 + TF_2 + ... + TF_i + ... + TF_m}{nm} + \frac{1}{2n} \tag{2}$$

Where m is the number of faults; n represents the number of test cases, and TF_i is the position of the first test case which detected the fault i. APFD$_C$ also measures the fault detection rate but can consider the cost of the test cases and the severity of the faults. The APFD$_C$ of a test suite is defined by the formula:

$$APFD_C = \frac{\sum_{l=1}^{m} (f_l \times (\sum_{j=TF_l}^{n} t_j - \frac{1}{2} t_{TF_l}))}{\sum_{l=1}^{n} t_l \times \sum_{l=1}^{m} f_l} \tag{3}$$

Where: f_l is the severity of the fault l and t_j is the cost (execution time) of the test j. When one of these values is not available it is still possible to use the metric, attributing the same value for the missing component of all test cases, i.e. the component is disregard since all test cases have the same value. In the evaluation of our approach, we used the APFD$_C$ only considering the cost because the faults' severity is not available for the study subjects.

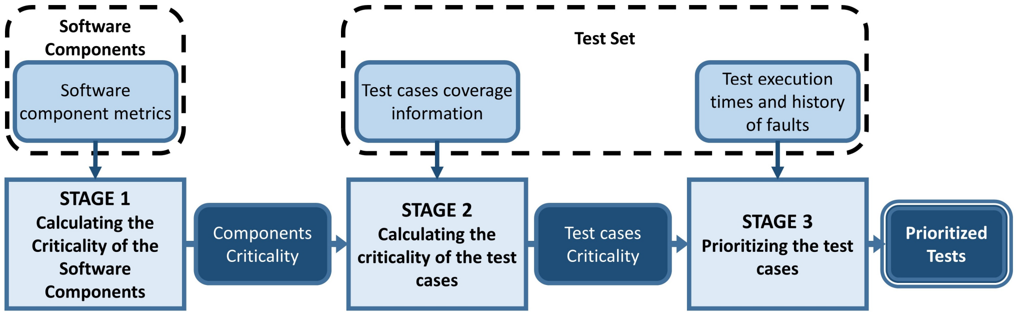
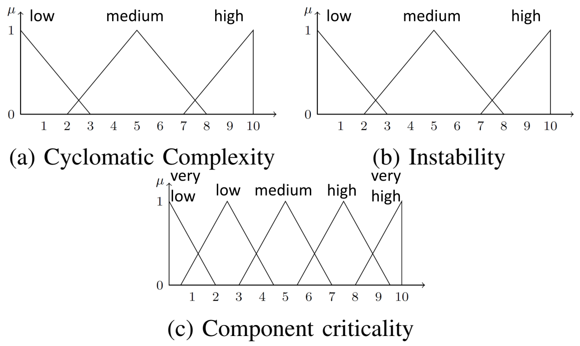

Software Components
Software component metrics
Test Set
Test cases coverage information
Test execution times and history of faults
STAGE 1 Calculating the Criticality of the Software Components
Components Criticality
STAGE 2 Calculating the criticality of the test cases
Test cases Criticality
STAGE 3 Prioritizing the test cases
Prioritized Tests

Fig. 1: Proposed approach



(a) Cyclomatic Complexity (b) Instability

(c) Component criticality

Fig. 2: Distribution of the fuzzy sets for the input variables (a) and (b), and the output variable (c)

TABLE II: Evaluation objects

Program	LOC	Functions	Test cases (pool)	Avg. cases/ suite	Selected suite (size)
print_tokens	402	18	4130	16	20
print_tokens2	483	21	4115	12	13
replace	516	22	5542	19	19
schedule	299	18	2650	8	11
schedule2	297	17	2710	8	10
space	6218	136	13585	155	155
tcas	138	9	1608	6	6
tot_info	346	9	1052	7	5

V. Empirical Evaluation

To evaluate the performance of the approach, we compared the obtained APFD and APFD$_C$ values with other versions of the approach, where the prioritization algorithm was changed. In this paper, we aim at answering the following research questions:

- **RQ1:** On average, how the results obtained with the approach perform when compared with the original order of the test suites, the optimal ordering (obtained through an exhaustive search) and the ordination using a greedy algorithm?
- **RQ2:** The results obtained with the approach are superior to the obtained with a random approach, at statistically and practically significant levels?

To conduct the evaluation, we used eight programs (Table II) written in C and obtained from the Software-Artifact Infrastructure Repository (SIR), a public repository of objects for experimentation in software engineering[1]. Each program contains seeded faults, a pool of test cases, and a set of 1000 small test suites, each one composed by a subset of the pool. The evaluation used one random test suite from each software system.

Our approach was implemented in the Java language. To obtain the inputs of the first step, the cyclomatic complexity of the code components (functions) of the subjects was calculated using CCCC (C and C++ Code Counter)[2]. The afferent and efferent couplings were obtained by analyzing the invocation tree of the components, generated using Scitools Understand[3]. The fuzzy system was implemented using jFuzzyLogic [19]. The defuzzification method chosen was the *Center of Area* [20].

In the second step, we used gcov to obtain the coverage percentage of the functions by each test case. This tool works together with *gcc*[4], a C compiler. In the third step, the MMAS algorithm had its α and β values set to 1, while the configured pheromone evaporation rate was 5%. The APFD$_C$ metric uses the cost (execution time, in milliseconds) of the test cases. To obtain these values, we executed the test cases and extracted the information from the execution logs. The evaluation objects do not have fault severity information. So, this parameter was set to 1 to all faults.

A. Approach tuning

To tune the parameters of the MMAS algorithm, we tested many configurations which would impact the results of the ACO algorithm, by changing the number of agents and the number of algorithm iterations. We also analyzed the contribution of the heuristic and pheromonal information and options for the heuristic function used in the building of the solution. In the initial experimentation, the approach was executed 100 times for each configuration. The object used in this step was the program space, due to its expressive quantities of code lines, test cases, and seeded faults. The heuristic used by the MMAS algorithm, except when indicated otherwise, is expressed by $(TC_j * F_j)/T_j$, where TC_j

[1]http://sir.csc.ncsu.edu
[2]https://sourceforge.net/projects/cccc/
[3]https://scitools.com/
[4]gcc.gnu.org

is the criticality (importance of execution) of the test case j; T_j is the execution time (cost) of the test case j; and F_j is the number of faults detected by the test case j (the fault detection potential based on the execution history).

To compare the different configurations we employed the Kruskal-Wallis test ($\alpha = 0.05$), because the data does not follow a normal distribution (to verify the normality we used the Shapiro-Wilk test, $\alpha = 0.01$). In the following subsections, the results obtained in the tuning step are presented.

1) Number of agents: To check the number of agents to be used in the statistical evaluation, the approach was executed using 10 configurations where the number of agents varied from 10 to 100, with a fixed number of iterations in the MMAS algorithm (3000). In the results (Table III) each intersection of a column and a row registers a p-value, which shows the statistical significance regarding the difference between the number of agents represented in the column and row labels. For the APFD metric, starting from 30 agents, the increase of 10 in the number of agents became less and less significant in terms of results, but the improvement in the results still can be observed when comparing to the other lesser quantities. For example, the statistical difference obtained using 40 agents instead of 30 is not significant, but when comparing the use of 40 and 20 agents, the observed statistical difference is significant. Based on the results obtained in the different configurations, on the next steps of the experimentation 80 agents will be used, due to the non-significant statistical difference to the immediately smaller quantities (70 and 60 agents).

TABLE III: Comparison between different numbers of agents - APFD

	10 ag.	20 ag.	30 ag.	40 ag.	50 ag.	60 ag.	70 ag.	80 ag.	90 ag.
20 ag.	<0.001	-	-	-	-	-	-	-	-
30 ag.	<0.001	<0.001	-	-	-	-	-	-	-
40 ag.	<0.001	<0.001	1	-	-	-	-	-	-
50 ag.	<0.001	<0.001	1	0.288	-	-	-	-	-
60 ag.	<0.001	<0.001	0.052	0.002	1	-	-	-	-
70 ag.	<0.001	<0.001	0.003	<0.001	0.521	1	-	-	-
80 ag.	<0.001	<0.001	<0.001	<0.001	0.019	1	1	-	-
90 ag.	<0.001	<0.001	<0.001	<0.001	<0.001	0.141	1	1	-
100 ag.	<0.001	<0.001	<0.001	<0.001	<0.001	0.0003	0.035	0.694	1

2) Number of iterations: To define the number of iterations of the ACO algorithm, the approach was executed using 10 different configurations. In every configuration, there were used 80 agents and a variable number of iterations - 1000 to 10000 (Table IV). From 4000 iterations on, the increase of iterations became less and less significant in terms of results. Based on the results, we will use 6000 iterations on the next steps of the experiment since, from this number on, it was observed a non-significant statistical difference to the two anterior quantities (4000 and 5000 iterations).

3) Importance of the pheromonal and heuristic information: To check the importance of the pheromonal information to the MMAS algorithm, the results obtained with the approach were compared with the results of a variation of the approach which MMAS did not consider the pheromonal information. Table V shows the average, the median and the standard deviation of the results for APFD and APFD$_C$ metrics, and the statistic difference of the results with and without pheromone. The results evidenced that the absence of pheromonal information resulted in a decrease in the results for both metrics. To evaluate the importance of the heuristic information, the results obtained with the complete approach were compared to the values obtained without using the heuristic information. In the same way, the heuristic information also showed itself important in the algorithm, and its remotion represented a decrease in the results for both metrics.

TABLE IV: Statistical difference between the results for different numbers of iterations (APFD)

	1000	2000	3000	4000	5000	6000	7000	8000	9000
2000	0.030	-	-	-	-	-	-	-	-
3000	<0.001	0.045	-	-	-	-	-	-	-
4000	<0.001	<0.001	1	-	-	-	-	-	-
5000	<0.001	<0.001	0.058	1	-	-	-	-	-
6000	<0.001	<0.001	<0.001	0.092	1	-	-	-	-
7000	<0.001	<0.001	<0.001	0.053	1	1	-	-	-
8000	<0.001	<0.001	<0.001	<0.001	0.046	1	1	-	-
9000	<0.001	<0.001	<0.001	<0.001	0.000	0.599	0.533	1	-
10000	<0.001	<0.001	<0.001	0.104	1	1	1	1	0.508

TABLE V: Analysis of the pheromone and heuristic information importance

	APFD			**APFD$_C$**		
	complete approach	without pheromone	without heuristic	complete approach	without pheromone	without heuristic
Avg.	96.86	95.15	95.87	97.05	95.65	95.96
Median	96.82	95.08	95.84	97.03	95.58	95.93
Std. Dev.	0.004	0.003	0.003	0.002	0.003	0.004
p-value		$<$ **0.001**	$<$ **0.001**		$<$ **0.001**	$<$ **0.001**

4) Heuristics evaluation: On the last step of the pre-experimentation, some heuristic options to build the solutions on the MMAS algorithm were evaluated (Table VI). The values employed in the heuristics were the criticality TC_j (to input in the solutions the execution importance of every test case j), the execution time T_j (to benefit faster test cases) and the number of faults detected by the test cases F_j (that offers evidence of the real capacity of fault detection).

- Heuristic 1 (H$_1$): TC_j/T_j
- Heuristic 2 (H$_2$): $(TC_j * F_j)/T_j$
- Heuristic 3 (H$_3$): F_j
- Heuristic 4 (H$_4$): $TC_j * F_j$
- Heuristic 5 (H$_5$): TC_j

It is possible to notice that the use of the fault history in the heuristics had a positive influence in the final result, and heuristics 2, 3 and 4 performed better at statistically significant levels. As expected, the execution time of the test cases did not present significant effects to the APFD metric. We can check this when comparing heuristics 1 and 5, or 2 and 4.

TABLE VI: Analysis of the statistic differences between the results for the heuristics

APFD					
Heur.	H_1	H_2	H_3	H_4	H_5
Median	95.35	96.82	96.95	96.84	95.27
H_2	<0.001	-	-	-	-
H_3	<0.001	<0.001	-	-	-
H_4	<0.001	1	1	-	-
H_5	1	<0.001	<0.001	<0.001	-

APFD$_C$					
Heur.	H_1	H_2	H_3	H_4	H_5
Median	95.82	97.03	97.14	97.05	95.32
H_2	<0.001	-	-	-	-
H_3	<0.001	0.939	-	-	-
H_4	<0.001	1	<0.001	-	-
H_5	<0.001	<0.001	<0.001	<0.001	-

For the metric APFD$_C$, the importance of considering the costs becomes evident when we observe that heuristic 1 performed better than heuristic 5. Heuristics 2 and 4 did not present a significant difference between them, but they did not also perform differently at a statistically significant level from heuristic 3. This can be due to the closeness to global optima offering low opportunity for improvement.

The use of the fault history alone (heuristic 3) brought a significant gain to the results. This was expected since the evaluation metrics are related to the fault-proneness. However, the fact that a test case found faults in a previous version does not mean it will keep finding faults. So, we opted by heuristic 2 in the approach, since it considers other information, and used heuristic 3 results as reference values.

B. Evaluation results

To evaluate the proposed approach, it was executed 1000 times for every evaluation object. Based on the previous tuning step, the prioritization algorithm (MMAS) employed 80 agents, 6000 iterations, and heuristic $(Test\ Criticality * Faults\ detected\ by\ the\ test)/Execution\ Time)$. We obtained also, for reference, the APFD and APFD$_C$ values of the original ordering of the test cases (non-ordination), the ordination employing a greedy technique and the optimal ordering, obtained by an exhaustive search.

The greedy technique and the optimal ordering consist of variations of the approach, where the prioritization algorithm (3rd step) changes. In the greedy technique, the prioritized test suite was mounted by successively selecting tests evaluated using the heuristic $(TCj * Fj)/Tj$. In the optimal ordering, the prioritized suite was obtained through the execution of an exhaustive search algorithm. The use of these techniques aimed to offer reference values to analyze the average of the values obtained through the approach.

Moreover, the approach was submitted to a sanity check and had its results compared, in terms of statistical and practical significance to the ones obtained by a random search, also executed 1000 times for each object. The results followed a non-normal distribution, demanding the execution of non-parametric tests. So, the statistical difference between the results of the approach and the random search was obtained by using a Wilcoxon-Mann-Whitney test with Bonferroni correction. We calculated the effect size of our approach, using the Vargha-Delaney A measure. The values for this metric vary between 0 and 1, and the closer to 0.5, the more similar are the compared groups. Values higher than 0.5 indicate that the proposed approach presented better results than the random search. Table VII states the results of the approach in comparison to the other results.

We can see that the greedy algorithm generated equal or better results than the non-ordination for both metrics, despite having its results visibly different from the optimal value. The average of the results obtained with the approach showed themselves equal to or greater than the reference values, to all objects and metrics. To every object where the exhaustive search could be performed, the approach results reached the global optimal value. So, *on average, the results obtained with the approach performed better than the original order of the test suites, surpassing the ordination using a greedy algorithm and, when the verification was possible, reaching the optimal ordering **(RQ1)**.*

In the sanity check, when compared to the random search, the approach showed superior results at statistical and practical levels (effect size). The results suggested, so, that our approach is capable to support the development team in the execution of the regression testing, being able to increase the fault detection rate of a set of test cases. So, *the results obtained with the approach for the studied subjects showed themselves higher than the obtained with a random approach, at statistically and practically significant levels **(RQ2)**.*

VI. Threats to Validity

Threats to internal validity: To avoid problems caused by errors in the instrumentation of the experimentation subjects, the information about the software components was collected by consolidated tools. The implementation of the fuzzy sets and rules can constitute a threat, but a set of empirical studies was performed aiming to find a configuration which could better reflect the specialists' knowledge about the knowledge domain.

Threats to external validity: The subject programs have small or medium size and simple fault patterns, which limits their representativity of complex software projects and their testing processes. To mitigate this threat, further experimentation must be performed, using objects more representative.

Threats to construct validity: A strong limitation of the evaluation metrics (APFD and APFD$_C$) is the fact that they do not consider the detection of faults that were already detected, information that could be useful to the debug team in the fault isolation.

This work has the following limitations:

- It was not possible to obtain the severity of the faults of the objects, and the experiments with the metric APFD$_C$ would have benefited from this information.
- In a practical application of the approach, it would be interesting to consider the relevance of the components

TABLE VII: Comparison of the results for the different techniques and analyses (APFD and APFD$_C$)

	APFD								APFD$_C$							
	Orig.	Greedy	Optimal	Rndm.	Apprch.	Std. Dev.	Stat. Dif.	A Measure	Orig.	Greedy	Optimal	Rndm.	Apprch.	Std. Dev.	Stat. Dif.	A Measure
print_tokens	82.50	96.42	-	76.12	**96.42**	-	<0.001	0.857	82.50	96.66	-	76.12	**96.66**	-	<0.001	0.857
print_tokens2	73.07	88.46	96.15	68.17	**96.15**	-	<0.001	0.923	73.07	88.46	96.15	68.17	**96.15**	-	<0.001	0.923
replace	83.47	83.87	-	49.84	**97.36**	-	<0.001	0.974	52.77	63.88	-	49.89	**97.22**	-	<0.001	0.974
schedule	54.54	63.63	95.45	59.83	**95.45**	-	<0.001	0.958	54.54	63.63	96.01	59.88	**96.01**	-	<0.001	0.958
schedule2	24.99	35.00	95.00	49.07	**95.00**	-	<0.001	1	16.66	38.88	94.44	49.07	**94.44**	-	<0.001	1
tcas	54.62	71.29	84.25	51.30	**84.25**	-	<0.001	0.996	59.09	77.27	83.33	51.36	**83.33**	-	<0.001	0.996
totinfo	67.14	67.14	81.42	54.55	**81.42**	-	<0.001	0.895	68.36	68.36	84.19	54.64	**84.19**	-	<0.001	0.895
space	83.47	83.87	-	83.07	**95.62**	0.34	<0.001	1	83.39	83.87	-	83.07	**95.76**	0.003	<0.001	1

in the calculation of the criticality of the software components, as made in Silva et al. [3].

- The comparison of the approach with other works was planned, however, it was not possible to extract some information in the subjects used in these papers or replicate the respective proposed approaches.

VII. CONCLUSION AND FUTURE WORKS

In this paper, we proposed an approach inspired by the proposal of Silva et al. [3] and extending [4]. Some contributions are related to improvements in the experimentation and validation of the approach, the tuning of parameters (number of agents, importance of the heuristic and pheromone information) and the choice of the heuristic for the ACO prioritization algorithm, to find a configuration that permits to reach better results without increasing excessively the computational cost of the approach.

The experimentation employed real programs from a public repository. The results of the approach were compared to results from a random approach, and we got evidence of the applicability of the approach. We also generated reference values for the subject programs using a greedy approach, an exhaustive search (when possible) and the unchanged test suites. In average, the results of the approach were greater than these reference values, reaching optimal values when the verification was possible. So, it became evident that the approach is applicable and efficient to regression testing environments.

As future work, we are working on an approach to automate relevance measurement, which will enable the usage of component relevance in the prioritization process. It is also relevant to perform complimentary studies about the relationship between software metrics and fault-proneness, aiming to improve the calculation of the criticality. Other optimization techniques, such as BrainStorm Optimization and Particle Swarm Optimization, are being studied as effective alternatives to the ACO algorithm. The approach could also benefit from the creation of a metric to evaluate the solutions based on the distribution of the test cases' criticality.

REFERENCES

[1] Pressman, R.: "Software Engineering: A Practitioner's Approach", 15th ed. New York, McGraw-Hill Higher Education, 2015

[2] Yoo, S., Harman, M.: "Regression Testing Minimization, Selection and Prioritization: a Survey", Software Testing, Verification and Reliability, vol. 22, no. 2, pp. 67-120, Chichester, UK. John Wiley and Sons Ltd. 2012

[3] Silva, D., Oliveira, P. A., Rabelo, R., Campanhã, M., Neto, P. S., Britto, R.: "A Hybrid Approach for Test Case Prioritization and Selection". IEEE Congress on Evolutionary Computation (CEC), Vancouver, BC, 2016, pp. 4508-4515. 2016

[4] Silva, D., Rabelo, R., Neto, P. S., Lima, G., Oliveira, P. A., Britto, R.: "An Approach for Test Case Prioritization Based on the Fault Detection History". Search-Based Software Engineering Workshop (WESB), 2017

[5] Zadeh, L. A.: "Fuzzy Logic = Computing With Words", IEEE Trans. Fuzzy Systems, vol. 4, no. 2, pp. 103-111, May 1996

[6] Rajendrani Mukherjee K. Sridhar Patnaik.: "A Survey on Different Approaches for Software Test Case Prioritization". Journal of King Saud University - Computer and Information Sciences, 2018

[7] Dorigo, M., Bonabeau, E., Theraulaz, G.: "Ant Algorithms and Stigmergy", Future Generations Computer Systems, vol. 16, no. 8, pp. 851-871, 2000

[8] Mamdani, E. H.: "Application of Fuzzy Logic to Approximate Reasoning Using Linguistic Synthesis", IEEE Transactions on Computers, vol. C-26, no. 12, pp. 1182-1191, Dec. 1977.

[9] Hao, D., Zhang, L., Zang, L., Wang, Y., Wu, X., Xie, T.: "To be Optimal or not in Test-Case Prioritization", IEEE Transactions on Software Engineering, vol. 42, no. 5, pp. 490-505, 1 May 2016.

[10] Elbaum, S., Malishevsky, A. G., Rothermel, G.: "Test Case Prioritization: A Family of Empirical Studies", IEEE Transactions on Software Engineering, vol. 28, no. 2, pp. 159-182, Feb. 2002.

[11] Kim, S., Baik,J.: "An Effective Fault Aware Test Case Prioritization by Incorporating a Fault Localization Technique". Proc. IEEE International Symposium on Empirical Software Engineering and Measurement, Bolzano-Bozen, Italy, 2010

[12] Gao, D., Guo, X., Zhao, L.: "Test Case Prioritization for Regression Testing Based on Ant Colony Optimization". 6th IEEE International Conference on Software Engineering and Service Science (ICSESS), Beijing, pp. 275-279, 2015

[13] Suri, B., Singhal, S.: "Understanding the Effect of Time-Constraint Bounded Novel Technique for Regression Test Selection and Prioritization", International Journal of System Assurance Engineering and Management, vol. 6, no. 1, pp. 71?77, March 2015.

[14] Acharya, A. A., Khandai, S., Mohapatra, D. P.: "A Novel Approach for Test Case Prioritization Using Business Criticality Test Value", Int. Journal of Computer Applications, vol. 46, no. 15, pp. 1-8, 2012

[15] Li, Z., Harman, M., Hierons, R. M.: "Search Algorithms for Regression Test Case Prioritization", IEEE Transactions on Software Engineering, vol. 33, no. 4, pp. 225-237, April 2007.

[16] McCabe, T. J.: "A Complexity Measure", IEEE Transactions on Software Engineering, vol. SE-2, no. 4, pp. 308-320, Dec. 1976.

[17] Martin, R. C.: "Agile Software Development: Principles, Patterns, and Practices", Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003

[18] Sttzle, T., Hoos, H. H.: "Max-Min Ant System", Future Generation Computer Systems, vol. 16, no. 8, Pages 889-914, 2000.

[19] Cingolani, P., J. Alcalá-Fdez: "jFuzzyLogic: a Java Library to Design Fuzzy Logic Controllers According to the Standard for Fuzzy Control Programming", Int. Journal of Computational Intelligence Systems, vol. 6, pp. 61-75, 2013.

[20] Ross, T. J.: "Fuzzy Logic with Engineering Applications", John Wiley & Sons, 4th Ed., 2016