Multi-Objective Hill Climbing for Automated RESTful API Test Case Generation

Jeroen Kappé , Mitchell Olsthoorn , Annibale Panichella
Delft University of Technology

Abstract

RESTful APIs tend to be difficult to manually write tests for. To help developers with this tedious task, a tool called EvoMaster has already been developed, to aim to automate the generation of test cases for RESTful APIs. The automation of test cases can be modeled as a multi-objective optimization problem. The existing tool EvoMaster has already implemented some global evolutionary search algorithms to solve this problem. In the context of RESTful APIs however, to this date, little attention has been given to a much simpler search algorithm: hill climbing. In contrast, hill climbing has been studied both inside and outside the context of search-based testing, and has found to be beneficial in a range of applications. The goal of this paper is to investigate the application of hill climbing within the context of testing RESTful APIs. The paper proposes a new local algorithm and compares its performance in the empirical study. Results comparing the local algorithm with the evolutionary counterpart show that the state of the art global algorithm is still superior. Although the empirical study also indicates that the novel hill climbing algorithm is able to cover a significant selection of branches, which the global one cannot.

1 Introduction

RESTful APIs are now common in online web services. They are widely used to exchange data between external web services over a network. However, testing RESTful APIs poses some challenges, since it involves HTTP input and output, which is hard to replicate [5]. This is problematic because software testing serves as an important validation tool for ensuring the reliability of the software [1]. Over the last decade, researchers have proposed various techniques to automate the test generation process, including for RESTful APIs. In particular, a tool called EvoMaster [6] has been created to automatically generate test cases for RESTful APIs.

Generating test cases is a multi-objective optimization problem. This problem is undecidable, but luckily, approximate optimal solutions are often good enough [13]. Metaheuristic search algorithms use heuristics to find these approximations. They can best be viewed as strategies ready for adaption for specific optimization objectives. Even in the context of search-based software testing, these objectives can vary greatly. In this paper, we focus on the objective to maximize the branch coverage of the system under test (SUT).

EvoMaster has already implemented the following global search optimization algorithms: Whole Test Suit (WTS) [9], Many-Objective Sorting Algorithm (MOSA) [14] and Many Independent Objective Algorithm (MIO) [4]. These Evolutionary Algorithms (EA) are inspired by nature and use concepts such as mutation and crossover to search for optimal solutions. A downside of this class of algorithms is that they generally require a long time (low search budget) to achieve acceptable results. To illustrate this, it is recommended to run EvoMaster up to 24 hours to achieve proper results in some cases. However, time and resource constraints are real-life challenges faced in both academia and the industry. So for some, the expensive evolutionary algorithms might be too time-consuming.

There also exist much simpler local search algorithms, such as the Hill Climbing (HC) algorithm, which are much faster in general. Interestingly, Arcuri [4] speculated that the HC approach might outperform existing evolutionary algorithms in low search budget scenarios [4]. In unit-testing, similar studies have already been conducted in this direction. In some scenarios, HC has been found to outperform evolutionary algorithms in terms of efficiency, [10]. But whether HC can be beneficial for system-level testing, within the context of RESTful APIs, is still an open question.

The goal of this paper is to shed light on how a hill-climbing approach can be successfully used in automated testing for RESTful APIs. The paper contributes to designing and implementing a novel local search algorithm in EvoMaster. On top of that, an empirical study comparing the performance of the new algorithm has been done. In terms of efficiency, the local algorithm was not able to exceed the current state of the art MIO algorithm. Nonetheless, it appears to be better in searching certain areas of the search space.

The remainder of this paper is organized as follows. Section 2 explains background information to understand the content of subsequent sections. Section 3 presents the novel many-objective hill climbing algorithm. Section 4 describes the design of the empirical study and reports the achieved results. Section 5 discusses the ethical aspects and reproducibil-

ity of the research. Section 6 and 7 wraps up with a discussion and conclusion of the study.

2 Background

This section covers background information required to know in order to fully understand the design and context of the newly proposed algorithm.

2.1 Testing RESTful APIs

Test cases targeting RESTful APIs consist of one or more HTTP calls. Each HTTP message consists of an operation together with extra metadata and additional payload data. This allows for a stateless communication to, for example, retrieve, update, or delete resources. On top of that, REST represents a style of additional constraints to make sure HTTP messages follow the HTTP semantics [7]. The components of an HTTP message consists of an HTTP verb, HTTP headers, path parameters, query parameters, and body payloads. The content of all of these components determines how "good" a test case is and is handled by EvoMaster [5]. In the next section, we will formulate what it means for a test case to be "good", according to the fitness function. Furthermore, the order of the HTTP calls also matters.

2.2 Search-based software testing

Formally the multi-objective optimization for maximizing branch coverage can be formulated as minimizing the fitness function for each branch. The fitness function consists of a metric called branch distance [15]. The branch distance represents how "close" the test case came to cover the condition of the branch. For example, if a branch is covered when a value is zero. For any input, the more the input deviates from the value zero, the higher the branch distance becomes. In this case, neither value 20 nor 1,000 would solve the constraint, but since 20 is closer, it has a relatively better fitness. This fitness function is used to guide the search for optimal test cases when evaluating candidate solutions. The primary goal is to maximize the coverage of the SUT. Second, one would like to minimize the total number of tests achieving this coverage goal as well [4].

The simplest approach for generating test cases is searching randomly [3]. In this approach, the content and order of each HTTP component are chosen at random. However, even for simple programs, there are a lot of possible values to choose from. Therefore, the performance of this approach is quite bad in cases where a branch has a small input domain fulfilling its condition since it is very unlikely that the randomly chosen value falls in the small range. Besides, random testing aims to generate as many test cases as possible, contradicting the second objective to minimize the size of the overall test suite.

In the case of randomly generating test cases, we don't use the guidance of the fitness functions in searching the optimal solution. However, far better results can be achieved when this guidance is used. Global search algorithms use this guidance and search the whole fitness landscape to search for the most optimal solution.

2.3 Local Search Optimization

In contrast, local search algorithms only search a part of the fitness landscape, one popular local search algorithm is hill climbing. Hill climbing was originally used for integers only. Hill climbing starts with an initial solution, randomly chosen from the search space. The neighbors of this solution are evaluated and the fitness of these are compared to the fitness of the initial solution. When a neighbor yields to a better fitness, this solution will be the new current solution, and its neighbors are again evaluated and compared. This process continues until a point is reached where no neighbors give an improvement in fitness.

During the evaluation of neighbors, it is either possible to take a random better neighbor, without first evaluating all other neighbors. This approach is called "stochastic hill climbing". Another approach is to evaluate all neighbors and take the best among them, this is called "steepest ascent". The name "hill climbing" is derived from the visual similarities the recursive moves have with climbing a hill on the fitness landscape, as can be visualized in a 2D or 3D plot.

The selection of the starting point is crucial since it determines what neighbors will be evaluated in subsequent moves, as only the local space will be searched. Therefore, it might happen that the final solution derived from the hill climbing search is not the best possible solution existing in the search space. This is a well-known limitation from which local search algorithms suffer. Two considerations can be made to try to overcome this weakness. First, the hill climbing process can be repeated several times, each time with a new randomly chosen starting point. Second, a less restricted search approach can be chosen. In this way, worse neighbors are accepted by a random probability, this is called simulated annealing. Both considerations only reduce the chance of getting stuck at local optima at best. Just like for fitness functions, the concept of neighbors is also problem specific. In our context, each solution represents a test case. The neighbors are determined by making exploratory moves on the parameters and the payloads.

One of the first approaches to use hill climbing in search-based software testing is the Altering Variable Method (AVM) [12]. AVM works on a vector of variables. Each variable is locally optimized using the hill climbing approach, without considering the other variables. When a local optimum is found for a specific variable, the next variable is chosen. After a full iteration on the variables, AVM loops back to the first one. This continues until no further improvements can be made in a cycle. In the next section, we will explain how we slightly modify this approach to best fit in the context of our multi-objective optimization problem. Although this algorithm is already quite old, it stood the test of time. Harmann and McMinn [10] have concluded that this simple approach is more efficient in some applications than more complex EA alternatives since it is less costly.

One question we could ask is, when can we expect a local approach to be superior to a global EA one, and under which circumstances can we expect the opposite? This depends on how to fitness landscape looks like. Harman and McMinn already performed an extensive theoretical analysis [10] to answer this question in the context of search-based

software testing on structural testing in general. To answer this, they used an already quite old concept called the "Royal Road" properties [8] a fitness landscape can have. The absence of these properties means the crossover of an EA, hence an EA in general is likely to not perform well, and as a result, a much simpler local HC approach is expected to be better. To put it in simpler terms, an EA algorithm is expected to perform better when the search space contains a lot of local optima, since local algorithm suffer greatly in this case, as previously described. Finding out to what degree these Royal Road properties are present in RESTful APIs is out of the scope of this paper, but it is important to realize that the type of fitness landscape significantly determines what search approach works best.

3 Algorithm

This section describes the novel hill climbing algorithm which has been implemented in EvoMaster. In particular, section 3.1 covers the needed extensions to be able to successfully handle all the data types possibly seen in an HTTP component. Since recall that the original hill climbing technique only handles integers. Next, we will cover how in turn the hill climbing technique is used as a many-objective branch coverage algorithm

3.1 Extension for data types

To successfully apply the HC technique in the domain of HTTP requests, we need to extend the original approach to support all the possible data types which can occur in the parameters and body content within an HTTP message. Below is listed all possible data types and is described how hill climbing moves are performed for these types.

Booleans and enumerations. For booleans, there always exists only one possible neighbor, this is the flipped opposite boolean. An exploratory move consists of only flipping the boolean value. Enumerations are represented as a collection of generic elements within EvoMaster. An exploratory move consists of going to either the left or right element within this collection. Although this collection can be arbitrarily big, the size is assumed to be small in practical scenarios.

Integer data types. The original hill climbing approach is used for searching integers, with an additional extension to speed up the search process. In the original approach, for any integer value, there exist two exploratory moves to explore, +1 and -1. However, this might mean that a lot of moves need to be performed before reaching a local optimum. To reduce the number of moves, for each successful move i, we increase or decrease the current value by a delta of 2^i. By doing this however, there is a large possibility of undershooting or overshooting the local optimum. When this happens, the delta is reset to 1, and moves are performed again starting from there. This simple extension greatly improves the performance in cases where the starting point is far away from the local optimum. For example when the starting point is 0, and the local optima is at value 1000. Reaching this without the extension requires 1000 moves to be performed. However, with using the extension, only 23 moves are required. Furthermore, a minimum and maximum limit are defined. This limits the search space, which improves performance.

Floating-point data types. For floating-point data types, the decimal values also need to be searched. Harman and McMinn [11], already studied an approach to accomplish this. This approach iterates through all the decimals of the value, and for each performs a local move similar to that of integers. However, this approach requires a lot of moves to be made, and its necessity to search through all the decimals reduces the efficiency. For this reason, a stochastic hill climbing approach is used. We apply the same approach as for integers for searching the significant digits. On top of that, the decimals are being explored by moving with a randomized delta, which follows a Gaussian distribution. A small standard deviation of 0.25 is used. This approach lacks the exactness compared to the one proposed by Harman and McMinn, but under the assumption that this exactness is often not needed, the approach is much more efficient.

Strings. Similarly, the search space for strings might become very large. There are too many possible neighbors to try to search in a reasonable amount of time. For this reason, local search on strings is done using stochastic hill climbing as well. This means that several random deletions, changes and insertions of characters are performed when moving to such random neighbor. The strings are randomly modified according to mutation, which is already in place for the existing evolutionary algorithms. The mutation on strings can best be interpreted as an educated randomization. On top of just randomly inserting, changing or deleting characters, mutation also considers specializations and only searches with valid characters. Specializations try to determine how a string is used in the SUT. For example, when a string is known to represent a date, mutations of this string are restricted to also be valid date representations.

Arrays. Within EvoMaster, arrays are represented as a collection of other data types. Exploratory moves on arrays work very similar to the original AVM approach. Meaning each variable in the array is locally optimized using hill climbing in isolation. Only one full iteration is performed.

Date and time data types. Date and time variables again use the AVM approach, this time allowing for multiple iterations. Dates are represented by three integers, which in turn represent the day, $month$ and $year$. Similarly, time variables are represented by the three integers $hour$, $minute$, $second$. An exploratory move is composed of applying integer hill climbing to each of these variables in isolation. So in principle, the same underlying moves are performed as for an array consisting of three integers.

Complex objects. An exploratory move for an arbitrary object consists of performing local search on each field variable in isolation, just like a single cycled AVM. For XML and JSON objects, this means performing search on the variables of all the elements. Maps are optimized in a similar way. Here keys are abstracted as fixed strings and keys are modifiable variables.

3.2 Archiving

Since we are dealing with optimizing multiple objectives (branches), a collection of promising test cases for each of these objectives should be kept. During the search, the algorithm keeps track of this collection in what is called the

archive. For every target branch, the corresponding performance of a test case on the fitness function determines whether this test case will be stored in the archive or not.

For a specific target, when a test case results in a fitness function of zero, this means that the target is covered by the test case. In this case, the test case is added to the archive. Later on, the test case might only be replaced by a shorter test case which also achieved a value of zero. A normalized fitness result of 1 represents the worst possible performance, so this test case will never be added. Test cases having at least some heuristic value, so somewhere between 0 and 1, will be stored in the archive, since it might be sampled later on.

When a test case is sampled from the archive, it is copied and HC is performed on its content. After that, the fitness is evaluated and in case it is improved for at least one branch, it is stored back to the archive for the branches for which it has seen an improvement.

Feedback Direction Sampling. When searching, it is important to invest our time wisely. As an example, one strategy is to invest all our time in covering a small group of branches with complex predicates which are therefore hard to cover. Instead, we could also spend that time on searching easier targets, which could be much more. The latter would be a more wise decision under time constraints. In the end, we are only interested in maximizing the amount of covered branches, not in maximizing the portion of complex ones.

Furthermore, only test cases that actually cover branches are of real value in the end, while test cases that only come really close are not. So for these reasons we have to be smart about what objective(s) we spend valuable time solving on. We do this with a technique called Feedback Directed Sampling, which has been proposed in the MIO algorithm. Full design details can be studied in the corresponding paper from Arcuri, [4]. Using this technique, we first invest our resources into covering the easy branches. After these targets are covered, we invest time in covering the more complex ones.

3.3 Exploration and exploitation trade-off

Similar to how we should make smart decisions on what to sample from the archive, wise choices should also be made to determine the degree of exploring and exploiting in the search space. Exploiting a small region in the search space refers to investing as much time as needed to try to locally optimize that region. This might result in a waste of time when these objectives might be infeasible or very hard to cover. On the other hand, we could also invest our time in exploring the whole search space as much as possible, and neglect performing hill climbing optimization as much as possible. This extreme describes the randomized approach.

To achieve the best possible solution under time constraints, a well-balanced exploration and exploitation search should be performed. In our algorithm, an emphasis is placed on exploring the search space at the beginning of the search. Later on, the algorithm tries to focus on optimizing just a selection of targets, which it has stumbled upon during its exploration. Below is reported how the balance is adapted over time in several different ways.

Random sampling vs archive sampling. Test cases are either randomly sampled or sampled from the archive, which

Algorithm 1: Many-Objective Hill Climbing Algorithm

```
Input  : Probability of random sampling P_r,
         Probability of structure mutation P_m,
         Stopping condition C
Output: Archive of optimized individuals A
T ← SetOfRandomPopulations()
A ← {}
while ¬C do
    if P_r > rand() then
        t ← RandomIndividual()
    else
        t ← SampleIndividual()
        if P_m > rand() then
            MutateStructure(t)
        end
        t ← HillClimbingSearch()
    end
    AddIfNeeded(A, t)
end
return A
```

Algorithm 2: Hill Climbing Search

```
Input  : Test case individual t,
         Stopping condition C,
         Maximum number of searches avmDepth,
         Probability of random reset P_r
Output: An optimized test case derived from t
if HasNoVariables(t) then
    return t
end
current ←t
for Variable in current.variables do
    for 1...avmDepth do
        if C then
            break
        end
        n ← GetBestNeighbor(t)
        if IsBetter(current, n) then
            if shouldContinueSearch then
                break
            end
            if P_r > rand() then
                randomize(Variable)
            end
            return current
        end
        current = n
    end
end
return current
```

stores promising test cases. To allow for exploration at the beginning of the search, random sampling is performed with a high probability. The probability of random sampling reduces linearly over time, so as the search continues, the chance of sampling from the archive and optimizing those increases.

AVM search depth. In the original AVM technique, an iteration through all the variables is performed, and only when a local optimum is found for a variable, the next one will be searched. In cases of complex branches, this means a lot of local moves are performed for each variable. To allow for exploration at the beginning of the search, a maximum number of neighbor comparisons has been set. This means that during the iteration, the next variable is selected either when the current variable is at a local optimum or when the number of times a new neighbor has been compared exceeds this maximum. At the beginning of the search, the maximum is low, and this is linearly increased over time, to allow for more exploitation as the search progresses. This maximum represents how in-depth we should focus on searching a single variable. To easily refer to this notion, we will call this maximum the "AVM depth" from now on.

Simulated Annealing. At the beginning of the search, simulated annealing is performed at a high probability. This probability reduces linearly over time. Simulated annealing allows us to explore the search area of worse neighbors. This is preferred at the start since exploring these neighbors might lead to more promising neighbors in the end.

Random reset. To still allow for partial explorations as the search progresses, a variable might be assigned a random new value whenever this variable has arrived at a local optimum. This allows the search to explore and possibly exploit a new region of the search space for this variable.

3.4 Proposed Multi-Objective Hill Climbing Algorithm

The techniques previously described are finally brought together in the newly proposed multi-objective hill climbing algorithm. At the beginning of the search or at a random probability (P_r), a randomly generated test case is sampled. After this, each component of the test case is optimized using the hill climbing technique, with a single AVM iteration. The main loop of the algorithm, which handles the sampling and saving to the archive can be seen in Algorithm 1.

The variable iteration representing the actual hill climbing technique is visualized in Algorithm 2. In subsequent matter, current neighbors are retrieved, evaluated and compared. This continues until the current test case performs better than any of its neighbors. At that point, there might still be an unexplored search space for the current optimized variable left. For example, it could happen in case of date variables, where only the day has been searched, while $month$ and $year$ are yet to be investigated. These two variables are then searched. In case the maximum AVM depth is reached, the search is stopped, any unexplored search space might be explored at a later stage in the search when the test is again retrieved from the archive.

To reduce the chance of getting stuck at local optima. The algorithm uses simulated annealing. For a random probability P_s, the algorithm returns a randomly chosen neighbor, instead of the best neighbor. Furthermore each time, all neighbors are evaluated and in case it improves the fitness of any not yet covered targets, it is stored in the archive. In that case, this neighbor might be chosen to be optimized for these specific targets later on in the search. Finally, since the structure of a test case determines its fitness, at a random probability, the structure is randomly mutated.

4 Empirical Study

To evaluate the performance of the proposed local algorithm, it was compared to the random search method, along with the MIO algorithm. We will only compare it with the MIO algorithm, and not to others such as MOSA, since the novel HC algorithm and MIO share common functionalities like the same use of archive and feedback direction sampling. Having these similarities makes comparing the two more interesting, since they only differ in how each optimizes a test case (locally vs. mutation). The comparison was tested on several different benchmark APIs derived from the EvoMaster benchmark project (EMB) [2]. These APIs are great for benchmark testing since they differ in complexity and function. For the comparison, the following benchmark APIs had been chosen:

- NCS: Artificial numerical example.
- Features-service: An API for managing products Feature Models.
- News: Small scale API used for a university-level course.

From this evaluation we aim to answer the following research questions:

> *RQ 1: How does the performance of the algorithm compare itself to the others in solving the branch coverage problem?*

For this comparison, we are interested in how many branches each algorithm can cover in a chosen time interval, and how quick (efficient) the coverage is increased during the search.

Hypothesis: The hill climbing algorithm will be more efficient at the beginning of the search compared to MIO, meaning it will achieve a higher branch coverage at the start. Furthermore, we speculate that as time progresses and the algorithm will get stuck in local optima and MIO will surpass the local algorithm. Still, the hill climbing algorithm should perform better than the random approach at all times.

> *RQ 2: What is the impact of the simulated annealing and AVM depth parameters on the branch coverage over time?*

To reduce the scope of the empirical study, only these two parameters are evaluated. Although the degree of random sampling and random resetting will also have an influence. However, their influence is expected to be less, and therefore they will not be studied.

Hypothesis: Both simulated annealing and AVM depth will influence the exploration/exploitation balance. More emphasis on exploration will result in higher coverage at the beginning of the search, but leaves the algorithm with difficulties

on fine-tuning test cases, so the coverage achieved in the end will be lower.

For simulated annealing, we expect that a start value of 0.25 and an end value of 0.0 will yield the best results. For AVM depth, it is expected that a start value of 1 and an end value of 10 will give the best balance of exploration and exploitation. 0.75 and 0.0 have been chosen for the probability of random sampling, and for the probability of random reset, the values are 0.0 and 0.50 for start and end. This configuration of parameters is used as default, for which a linear interpolation is used during the search for each parameter.

4.1 Experimental procedure

In the interest of answering the first research question, we executed a total of ten independent runs for each algorithm on each of the selected API benchmark. Each run takes 300 seconds (five minutes) since the benchmark APIs are small and the algorithms are expected to converge within this time. During the search, we computed the achieved branch coverage thus far. In the end, we compare the coverage over time to compare the efficiency as well as the achieved total coverage at the end.

For answering the second research question, a similar set up is used. We alternate different values for each parameter and executed five independent runs. While doing this, we keep the other parameters set to their default values. To simplify the study, we only change the starting value of simulated annealing and the ending value for AVM depth. For AVM depth, we alternate the ending value between 5, 10 and 15. Furthermore, the starting value of the probability of performing simulated annealing is alternated between 0.0, 0.25 and 0.50.

4.2 Results

Algorithms	Covered branches during search			
NCS	25%	50%	75%	100%
Random	101	104	104	104
MIO	142	184	152	153
HC	108	119	122	126
Features				
Random	5	5	5	5
MIO	11	11	11	11
HC	9	9	9	9
News				
Random	49	49	49	49
MIO	58	59	59	60
HC	50	51	52	52

Table 1: Average number of covered branches after the percentage of used search budget, for each of the algorithms on each selected benchmark project.

Table 1 shows all the retrieved results for the first research question. For four chosen timestamps during the search, the corresponding amount of covered branches for each algorithm is reported. Hence, the last column contains the results the algorithm was able to achieve during the whole five-minute run. Figure 1 shows the average coverage of branches

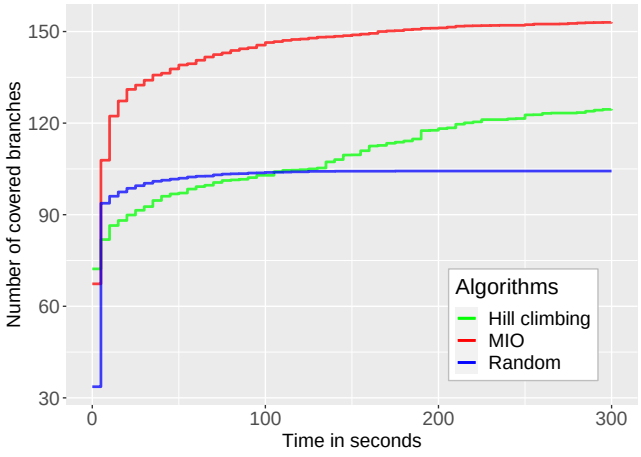

Figure 1: NCS - Average coverage over time for the HC, MIO and Random algorithm.

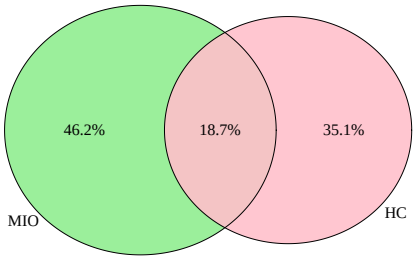

Figure 2: NCS - Distribution of covered branches by both MIO and HC. Green represents the branches covered by MIO, the red area represents the branches covered by HC.

during a 300 seconds run for the selected algorithms for the NCS project. Note that each of the algorithms, especially the random algorithm, seems to converge to their limit.

The results for the second research question are visualized in figure 4 and 5 for the NCS and Features benchmarks respectively. The coverage over time is plotted for each value of the parameters. Furthermore statistical properties are reported in table 2. There, the baseline represents the one using the default settings, as previously described. In addition, the p-value, applied on the final values, resulting from the Wilcoxon Test describing the statistical significance is also reported, as well as the Vargha-Delaney A_{12} value which describes the magnitude of the difference in performance compared to the baseline algorithm.

Figure 2 and 3 visualize the distribution of the covered branches by both MIO and HC in percentages, for the NCS and Features benchmark respectively. The green circle represents all the branches covered by MIO, while the red circle is the collection of covered branches by HC. The area laying inside the two circles represents the percentage of branches covered by both.

Parameter settings	Covered branches during search						
	25%	50%	75%	100%	p-value	A_{12}	Magnitude
NCS							
Baseline	108	119	122	126	-	-	-
Simulated annealing - 0.0	95	113	127	131	0.20	0.46	negligible
Simulated annealing - 0.50	104	113	122	125	0.85	0.45	negligible
AVM depth - 5	100	109	114	119	0.10	0.27	medium
AVM depth - 15	102	117	122	126	1.00	0.45	negligible
Features							
Baseline	8.6	8.8	8.8	8.8	-	-	-
Simulated annealing - 0.0	9.2	9.2	9.4	9.4	0.59	0.83	large
Simulated annealing - 0.50	9.2	9.8	9.8	9.8	0.37	0.88	large
AVM depth - 5	8.4	8.8	8.8	9.0	0.85	0.55	negligible
AVM depth - 15	9.2	9.4	9.4	9.4	1.00	0.85	large
Scout							
Baseline	50	51	52	52	-	-	-
Simulated annealing - 0.0	52	52	53	53	0.50	0.65	medium
Simulated annealing - 0.50	49	51	52	52	0.41	0.31	medium
AVM depth - 5	52	52	53	54	0.40	0.75	large
AVM depth - 15	51	51	51	52	0.37	0.47	negligible

Table 2: Average number of covered branches after the percentage of used search budget, for each different values of exploration/exploitation parameters. Furthermore the p-value from the Wilcoxon test, Vargha-Delaney A_{12} and effect magnitude values are included.

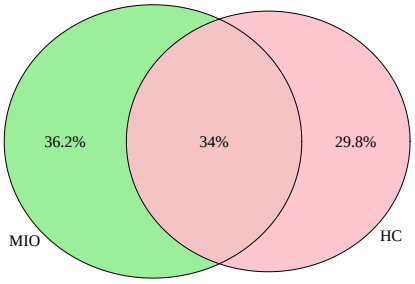

Figure 3: Features - Distribution of covered branches by both MIO and HC. Green represents the branches covered by MIO, the red area represents the branches covered by HC.

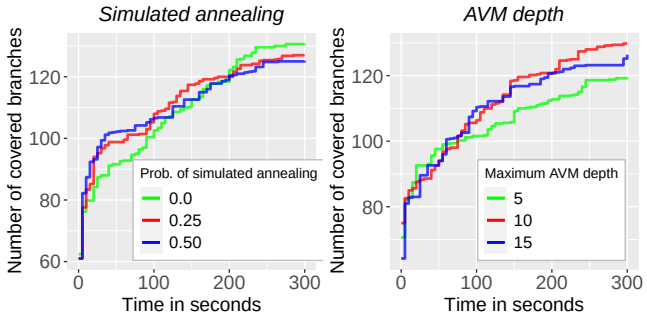

Figure 4: NCS - Coverage over time with different exploration/exploitation parameter values.

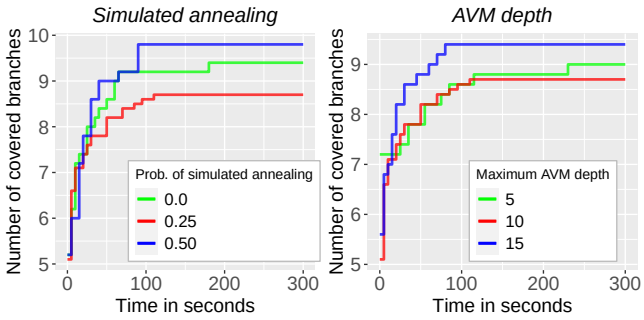

Figure 5: Features - Coverage over time with different exploration/exploitation parameter values.

5 Responsible Research

To minimize the threats to validity we executed each experiment multiple times. This is important since we are dealing with an algorithm based on randomness, meaning it does not give the same result back every time. Executing the experiment multiple times, and analyzing the average of these gives us more confidence about the validity of the results. While more runs mean more confidence, due to time constraints only ten runs for the first research question, and five for the second one were done.

Furthermore, we cannot prove that the implemented code is 100 percent bug-free. Of course we did our best to exclude any as much as possible and while the code is thoroughly tested, still the absence of bugs will never be a certainty.

The algorithm was made with the goal of being more efficient than the existing state of the art MIO algorithm. While solely looking at the achieved branch coverage, the hill climbing algorithm was not able to surpass the existing MIO approach, still no artificial or modified data was used to make

the algorithm shine more.

Science has always been about reproducibility. Without being able to reproduce an experiment, the attained results do not mean much. Luckily, EvoMaster is an open-source project, so reproducing the attained results is a matter of engineering effort for only the concepts discussed in the paper. While it would be better to make the discussed additions open source as well, still the amount of effort for these changes are minor compared to those of the whole EvoMaster project.

6 Discussion

Solely looking at the sum of covered branches we can conclude that hill climbing does not seem to be able to give an improvement compared to the existing MIO algorithm, since HC was not able to surpass MIO in terms of efficiency. Still, it is fortunately better than randomly testing, giving us more confidence that there are not any major errors happening.

There are three speculations we have about the performance of the novel hill climbing algorithm compared to the state of the art MIO algorithm.

First, the approach to tackle the multiple objectives might not be best suited for local search. Although the same archiving and feedback-directed sampling techniques are used as found in MIO, in which it shows good results, it might not be the desired technique for hill climbing. Instead, the use of preference sorting and dominance, as used in MOSA [14] might yield better results. However, we don't expect this to be the case since the used techniques are shown to be powerful in MIO. Luckily, combining the techniques seen in MOSA and local search, which are both already in place, is a matter of relatively low engineering effort.

Second, the genetic approach of MIO seems to be able to more efficiently search some data-types better than HC. Assumed is that the efficiency of stochastic local search, as used on strings, is quite low since it is just random guessing. This is unfortunate since strings have a strong presence in the evaluated benchmark projects and RESTful APIs in general. So although HC might be powerful when applied on numerical data-types, it lacks any structure on others. Recall that we use this approach in cases where the search space is too large to apply standard local search on. So we suspect MIO to be able to handle this vastness better than the local algorithm.

At last, although it is hard to visualize how the fitness landscape looks like [11], it might be that it contains too many local optima in this specific context. If this is the case, local search will never be an attractive option, no matter how it is implemented. Although we have not proven this to be the case, so it remains to be just a hypothesis.

However, there is a catch. While HC covered fewer branches than MIO after all, HC seems to more easily cover a certain area of the search better than the global counterpart. As pictured in figures 2 and 3, there seems to be a big distinction between what targets were covered by MIO and HC. Of all the covered branches by the two algorithms combined, roughly 35 percent and 30 percent, in the cases of NCS and Features respectively, were only covered by HC. So more or less a third of the search space was unreachable for MIO while in reach for HC. This is a big chunk of the search space.

An assumption made is that the shared percentage of branches of HC and MIO are relatively easy targets to hit, and are possibly also easily covered by the randomized algorithm. The extension of the green MIO circle not laying in the red cycle represents the part of the search space dominated by the MIO algorithm. This part most likely contains the Royal Road properties as previously discussed. But interestingly enough, the big red HC part of the search space is the area of the search space where the local approach dominates in searching more efficiently. This area probably contains a lot of numerical data-types, since local search is likely to perform better on those. This is in line with the retrieved results, since NCS contains a lot of numerical data-types and HC shows to cover a larger percentage of distinct branches (35%) in that case compared to the Features environment (30%). Overall the large mismatch between what branches were covered by what algorithm is very interesting, and indicates that both have an area in the search space for which it dominates the other algorithm in terms of search efficiency.

For the second request question, there is not any configuration of parameters that outperforms others all of the time. This follows the famous principle in computer science that there is no free lunch for a search algorithm on average applied on a class on problems. Instead, the optimization of the performance of each configuration is to be achieved by fine-tuning for each specific problem.

7 Conclusions and Future Work

In this paper, we have looked at the concepts of local search in the context of optimizing branch coverage for RESTful APIs, where previously only global evolutionary algorithms have been used. We implemented a new multi-objective hill climbing algorithm. We also carried out an empirical study, comparing the novel algorithm with the existing state of the art global MIO algorithm.

The performance in terms of efficiency is still better for the MIO algorithm compared to the novel local algorithm. This is likely due to the extensive size of the search space, as well as the potentially large number of local optima within it. On the bright side, the local competitor is able to search a large percentage of the search space more efficiently. Of the total amount of covered branches, around 30% was solely covered by the new local algorithm. The hill climbing method seems to be particularly useful in searching numerical data-types within this search landscape.

So while a standalone local approach did not surpass the performance of the current global state of the art algorithm, a smart combination of the two methods might yield an even better searching algorithm, one which is able to cover more branches. In the most optimistic scenario, when hill climbing is wisely used on its preferred search area, while MIO is used on the other parts of the search space, this combined approach might see a 30% percent increase in attained branch coverage. Future work will have to show how a local and global search approach is best to be combined in a hybridized algorithm. This hybridized approach is likely to be supreme in search-based software testing in the context of RESTful APIs in EvoMaster.

References

[1] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[2] A. Arcuri. Evomaster: Evolutionary multi-context automated system test generation. pages 394–397, 2018.

[3] A. Arcuri, M. Z. Iqbal, and L. Briand. Random testing: Theoretical results and practical implications. *IEEE Transactions on Software Engineering*, 38(2):258–277, 2012.

[4] Andrea Arcuri. Many independent objective (mio) algorithm for test suite generation. 2017.

[5] Andrea Arcuri. Restful api automated test case generation with evomaster. *ACM Trans. Softw. Eng. Methodol.*, 28(1), January 2019.

[6] Andrea Arcuri. Evomaster, 2020.

[7] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Irvine, 2000.

[8] MMS Forrest and J Holland. The royal road for genetic algorithms: Fitness landscapes and ga performance. 1991.

[9] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.

[10] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2010.

[11] Mark Harman and Phil McMinn. A theoretical empirical analysis of evolutionary testing and hill climbing for structural test data generation. page 73–83, 2007.

[12] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[13] Phil McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[14] A. Panichella, F. M. Kifetew, and P. Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, 2018.

[15] Joachim Wegener, André Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and software technology*, 43(14):841–854, 2001.