Test Suite Generation with the Many Independent Objective (MIO) Algorithm

Andrea Arcuri[a,b]

[a] *Westerdals Oslo ACT, Faculty of Technology, Oslo, Norway*
[b] *University of Luxembourg, Luxembourg*

Abstract

Context: Automatically generating test suites is intrinsically a multi-objective problem, as any of the testing targets (e.g, statements to execute or mutants to kill) is an objective on its own. Test suite generation has peculiarities that are quite different from other more regular optimisation problems. For example, given an existing test suite, one can add more tests to cover the remaining objectives. One would like the smallest number of small tests to cover as many objectives as possible, but that is a secondary goal compared to covering those targets in the first place. Furthermore, the amount of objectives in software testing can quickly become unmanageable, in the order of (tens/hundreds of) thousands, especially for system testing of industrial size systems.

Objective: To overcome these issues, different techniques have been proposed, like for example the Whole Test Suite (WTS) approach and the Many-Objective Sorting Algorithm (MOSA). However, those techniques might not scale well to very large numbers of objectives and limited search budgets (a typical case in system testing). In this paper, we propose a novel algorithm, called Many Independent Objective (MIO) algorithm. This algorithm is designed and tailored based on the specific properties of test suite generation.

Method: An empirical study was carried out for test suite generation on a series of artificial examples and seven RESTful API web services. The EvoMaster system test generation tool was used, where MIO, MOSA, WTS and random search were compared.

Results: The presented MIO algorithm resulted having the best overall performance, but was not the best on all problems.

Conclusion: The novel presented MIO algorithm is a step forward in the automation of test suite generation for system testing. However, there

are still properties of system testing that can be exploited to achieve even better results.

Keywords:
test generation, SBSE, SBST, multi-objective optimization, system testing

1. Introduction

Test case generation can be modelled as an optimisation problem, and so different kinds of search algorithms can be used to address it [1]. There can be different objectives to optimise, like for example branch coverage or the detection of mutants in the system under test (SUT). When aiming at maximising these metrics, often the sought solutions are not single test cases, as a single test cannot cover all the objectives in the SUT. Often, the final solutions are sets of test cases, usually referred as *test suites*.

There are many different kinds of search algorithms that can be used for generating test suites. The most famous is perhaps the Genetic Algorithms (GA), which is often the first choice when addressing a new software engineering problem for the first time. But it can well happen that on specific problems other search algorithms could be better. Therefore, when investigating a new problem, it is not uncommon to evaluate and compare different algorithms. On average, no search algorithm can be best on all possible problems [2]. It is not uncommon that, even on non-trivial tasks, simpler algorithms like (1+1) Evolutionary Algorithm (EA) or Hill Climbing (HC) can give better results than GA (e.g., as in [3]).

A major factor affecting the performance of a search algorithm is the so called *search budget*, i.e., for how long the search can be run, usually the longer the better. But the search budget is also strongly related to the tradeoff between the *exploitation* and *exploration* of the search landscape. If the budget is low, then a population-based algorithm like GA (which puts more emphasis on the exploration) is likely to perform worse than a single, more focused individual-based algorithm like HC or (1+1) EA. On the other hand, if the search budget is large enough, the exploration made by the GA can help it to escape from the so called local optima in which HC and (1+1) EA can easily get stucked in.

To obtain even better results, then one has to design specialised search algorithms that try to exploit the specific properties of the addressed problem domain. In the case of test suite generation, there are at least the following peculiarities:

- testing targets can be sought *independently*. Given an existing test suite, to cover the remaining testing targets (e.g., lines and branches), you can create and add new tests without the need to modify the existing ones in the suite. At the end of the search, one wants a minimised test suite, but that is a secondary objective compared to code coverage.

- testing targets can be strongly related (e.g., two nested branches), as well as being completely independent (e.g., code in two different top-level functions with no shared state).

- some testing targets can be *infeasible*, i.e., impossible to cover. There can be different reasons for it, e.g., dead code, defensive programming or the testing tool not handling all kinds of SUT inputs (e.g., files or network connections). Detecting whether a target is feasible or not is an undecidable problem.

- for non-trivial software, there can be a very large number of objectives. This is specially true not only for system-level testing, but also for unit testing when mutation score is one of the coverage criteria [4]. Traditional multi-objective algorithms are ill suited to tackle large numbers of objectives [5].

In this paper, we propose a novel search algorithm that exploits such characteristics and, as such, it is specialised for test suite generation (and any problem sharing those properties). We call it the Many Independent Objective (MIO) algorithm. We carried out an empirical study to compare the MIO algorithm with the current state-of-the-art, namely the Whole Test Suite (WTS) [6] approach and the Many-Objective Sorting Algorithm (MOSA) [7]. We also used random search as a baseline.

We carried out a series of experiments on a set of artificial software with different characteristics (clear gradients, plateaus, deceptive local optima and infeasible targets). In most cases MIO achieves higher coverage. However, artificial problems might not be fully representing the characteristics of real software. Therefore, we also carried out experiments on system testing of seven RESTful API web services using the open-source EvoMaster[1] tool.

[1] http://www.evomaster.org

Overall, MIO obtained the best results, but not on all of the seven web services. In these latter cases though, MIO still obtained the second-best results. Therefore, it is important to understand why this was the case, and what can be learned to propose novel MIO variants to improve performance even further.

To enable the replicability of this study, and to enable the use of MIO in other research and in industrial contexts, we provide the source code of all experiments and case studies as open-source on GitHub[2], currently the most popular open-source repository.

This article is an extension of a conference paper [8]. In particular, in this extension we analyze a further variant of how MIO can handle infeasible targets. Furthermore, the original [8] only had experiments on artificial examples and three tiny numerical functions, which are replaced here with the system testing of seven RESTful APIs.

This paper is organized as follows. Section 2 provides background information needed to understand the rest of the paper. The novel MIO algorithm is presented in Section 3. The empirical study is described in Section 4. Threats to validity are discussed in Section 5. Finally, Section 6 concludes the paper.

2. Background

2.1. Search-Based Software Testing

There has been a lot of research on how to automate the generation of high quality test cases. One of the easiest approach is to generate test cases at random [9]. Although it can be effective in some contexts, random testing is often not an effective strategy.

Among the different techniques proposed throughout the years, search-based software engineering has been particularly effective at solving many different kinds of software engineering problems [1], in particular software testing [10], with advanced tools for unit test generation like EvoSuite[3] [11, 12].

Software testing can be modeled as an optimization problem, where one wants to maximize the code coverage and fault detection of the generated test suites. Then, once a fitness function is defined for a given testing problem, a search algorithm can be employed to explore the space of all possible solutions (test cases in this context).

[2]https://github.com/EMResearch
[3]https://github.com/EvoSuite/evosuite

There are several kinds of search algorithms, where Genetic Algorithms (GAs) are perhaps the most famous. In a GA, a population of individuals is evolved for several generations. Individuals are selected for reproduction based on their fitness value, and then go through a crossover operator (mixing the material of both parents) and mutations (small changes) when sampling new offspring. The evolution ends either when an optimal individual is evolved, or the search has run out of the allotted time.

2.2. Whole Test Suite (WTS)

The Whole Test Suite [6] approach was introduced as an algorithm to generate whole test suites. Before that, typical test case generators were targeting only single objectives, like specific lines or branches, using heuristics like the *branch distance* and the *approach level* (as for example done in [13]).

In the WTS approach, a GA is used, where an individual in the GA population is a set of test cases. Mutation and crossover operators can modify both the set composition (i.e., remove or add new tests) and its content (e.g., modify the tests). As fitness function, the sum of all branch distances in the SUT is used. At the end of the search, the best solution in the population is given as output test suite. To avoid losing good tests during the search, the WTS can also be extended to use an *archive* of best tests seen so far [14].

2.3. Many-Objective Sorting Algorithm (MOSA)

The Many-Objective Sorting Algorithm (MOSA) [7] was introduced to overcome some of the limitations of WTS. In MOSA, each testing target (e.g., lines) is an objective to optimize. MOSA is an extension of NSGA-II [15], a very popular multi-objective algorithm. In MOSA, the population is composed of tests, not test suites. When a new target is covered, the test covering it gets stored in an archive, and such target is not used any more in the fitness function. A final output test suite is composed by the best tests found during the search and that are stored in the archive.

In NSGA-II, selection is based on ranks (from 1 on, where 1 is the best): an individual that subsumes many other individuals gets a better rank, and so it is more likely to be selected for reproduction. One of the main differences of MOSA compared to NSGA-II is the use of the *preference sorting criterion*: to avoid losing the best individuals for a given testing target, for each uncovered testing target the best individual gets the best rank (0 in MOSA), regardless of its subsuming relations with the other tests.

2.4. EvoMaster

EvoMaster[4] is an open-source tool written in a mix of Kotlin and Java. EvoMaster aims at generating system level test cases, using search-based techniques. EvoMaster currently focuses on web services, in particular RESTful APIs [16].

EvoMaster is composed of two main components: a *core* process responsible for the main functionalities (e.g., command-line parsing, search and generation of test files), and a *driver* process. This latter is responsible to start/stop/reset the SUT and instrument its source code, e.g., via automated bytecode manipulation, in a similar way of how unit test tools like EvoSuite [11] do. For example, you need to add probes in the bytecode to check which statements are executed, and also to define heuristics to help solving the predicates in the branch statement (e.g., the so called *branch distance* [17]).

EvoMaster generates test suites with the goal of optimising white-box, code coverage metrics (e.g., statement and branch coverage) and fault detection (e.g., HTTP 5xx status codes can be used in some cases as automated oracles). Each test case will be composed of one or more HTTP calls. The generated test files (e.g., using JUnit[5] and RestAssured[6] libraries) are self-contained, as using the EvoMaster driver as a library to automatically start the SUT before running the tests (e.g., in JUnit this can be done in a `@BeforeClass` init method).

Generating test cases for RESTful APIs is a complex tasks, with a very large search space. Not only one needs to decide how many HTTP calls to do, but for each call we also need to setup all the appropriate HTTP headers, specify the HTTP verb (e.g., POST or GET), URL path parameters, URL query parameters and HTTP body payloads. These latter could be arbitrarily complex (many fields with string and numerical values), representing data usually encoded in JSON or XML formats.

A REST endpoint is usually implemented with a function in a class (e.g., in Java). Server frameworks (e.g., Spring and JEE) will automatically call such function when a HTTP request is received (which function is called depends on the URI of the requested HTTP resource). The data in a HTTP

[4]http://www.evomaster.org

[5]http://junit.org/junit4/

[6]https://github.com/rest-assured/rest-assured

```java
@PUT
@Timed
@Path("{id}")
@Produces(MediaType.APPLICATION_JSON)
@UnitOfWork
public Tag update(@Auth @ApiParam(hidden = true) AuthResult authResult,
         @Context HttpServletResponse response,
         @PathParam("id") long id,
         Tag updatedTag) {
       doAuth(authResult, response, Permission.category_edit);
       Tag persisted = dao.read(id);

       persisted.setName(updatedTag.getName());
       persisted.setGroup(updatedTag.getGroup());

       dao.update(persisted);
       return persisted;
   }
```

Figure 1: Example of a simple RESTful endpoint from the *scout-api* SUT.

```java
@Test
public void test9() throws Exception {

   given().accept("*/*")
          .header("Authorization", "ApiKey administrator")
          .contentType("application/json")
          .body("{\"id\":-1116301531,
                \"group\":\"MGwNpDeS8y\",
                \"media_file\":{\"mime_type\":\"xLoD2NC72Ag\",
                       \"author\":\"4o9JkwgN7\"},
                       \"activities_count\":-1696622830}")
          .put(baseUrlOfSut + "/api/v1/categories/-5331650344483936669")
          .then()
          .statusCode(500);
}
```

Figure 2: Example of test generated by EvoMaster for the endpoint in Figure 1.

request (e.g., payload and headers) is automatically unmarshalled into the inputs of the endpoint's function (e.g., strings and data transfer objects).

Figure 1 shows a simple example of an endpoint from the *scout-api* SUT. In that endpoint, a PUT is handled to replace an existing resource. Such resource is first loaded from the database, and the updated values are then persisted back. Figure 2 shows an example of test generated by EvoMaster for such endpoint, using the RestAssured library. The payload is written in JSON, which will be unmarshalled into the `Tag` Java class when the HTTP request is handled by the server. Note that such test reveals a bug in the SUT, as the returned HTTP status is 500 (server error). When doing an update, the SUT does not check if the resource to update exists, leading to a null pointer exception when an invalid id is given as input. The correct behaviour in this case would had been to return a 404 status code.

3. The MIO Algorithm

3.1. Core Algorithm

Both WTS and MOSA have been shown to provide good results, at least for unit test generation [6, 14, 7]. However, both algorithms have intrinsic limitations, like for example:

- population-based algorithms like WTS and MOSA do put more emphasis on the exploration of the search landscape, which is not ideal in constrained situations of limited search budgets, like for example in system-level testing where each test case execution can be computationally expensive. Letting the user to tune the population size parameter is not a viable option, unless it is done automatically (but even then, it has side effects, as we will see in the empirical study).

- although once a target is covered it is not used any more for the fitness function, the individuals optimised for it would still be in the population. They will die out eventually after a few generations, but, until then, their presence in the population can hamper the search if those covered targets are unrelated to the remaining non-covered targets.

- in the presence of infeasible targets, some tests can get good fitness score (e.g., a close to 0 branch distance) although they will never cover those infeasible targets. Those not useful tests might end up taking over a large part of the population.

- there can be a very large number of objectives to cover, even in the order of hundreds of thousands (e.g., in the system-level testing of industrial systems). A fixed size population would simple not work well: if too small, then there would not be enough diverse genetic material in the first generation; if too large, not only convergence would be drastically slowed down, but also the computational cost could sky-rock (e.g., NSGA-II has a quadratic complexity based on the population size).

Algorithm 1 Many Independent Objective (MIO) Algorithm

Input: Stopping condition C, Fitness function δ, Population size limit n, Probability of random sampling P_r, Start of focused search F

Output: Archive of optimised individuals A

```
T ← SetOfEmptyPopulations()
A ← {}
while ¬C do
    if P_r > rand() then
        p ← RandomIndividual()
    else
        p ← SampleIndividual(T)
        p ← Mutate(p)
    end if
    for all k ∈ ReachedTargets(p) do
        if IsTargetCovered(k) then
            UpdateArchive(A,p)
            T ← T \ {T_k}
        else
            T_k ← T_k ∪ {p}
            if |T_k| > n then
                RemoveWorstTest(T_k,δ)
            end if
        end if
    end for
    UpdateParameters(F,P_r,n)
end while
return A
```

To avoid these limitations, we have designed a novel evolutionary algorithm that we call the Many Independent Objective (MIO) algorithm. In a nutshell, MIO combines the simplicity and effectiveness of (1+1) EA with a dynamic population, dynamic exploration/exploitation tradeoff and feedback-directed target selection. Algorithm 1 provides a high level pseudo-code of the MIO algorithm. The full details of the algorithm can be found (written in Kotlin) in the repository of the EvoMaster tool, in particular in the class `MioAlgorithm`.

The MIO algorithm maintains an archive of tests. In the archive, *for* *each* testing target we keep a different population of tests of size up to n (e.g, $n = 10$). Therefore, given z objectives/targets, there can be up to $n \times z$ tests in the archive at the same time.

At the beginning of the search, the archive will be empty, and so a new test will be randomly generated. From the second step on, MIO will decide to either sample a new test at random (probability P_r), or will choose (details later) one existing test in the archive (probability $1 - P_r$), copy it, and mutate it. Every time a new test is sampled/mutated, its fitness is calculated, and it will be saved in the archive if needed (details later). At this point, we need to define how tests are saved in the archive, and how MIO samples from the archive.

When a test is evaluated, a copy of it might be saved in 0 or more of the z populations in the archive, based on its fitness value. For each target, there will be a heuristic score h in [0,1], where 1 means that the target is covered, whereas 0 is the worst possible heuristic value. For example, if the heuristic is the branch distance d, this can be mapped into [0,1] by using $h = 1/(1 + d)$ (where $h = 0$ if a branch was never reached and so the branch distance d was not calculated).

For each target k, a test is saved in population T_k, with $|T_k| \leq n$, if either:

- if $h_k = 0$, the test is not added regardless of the following conditions.
- if the target is covered, i.e. $h_k = 1$, the test is added and that population is shrunk to one single individual, and it will never expand again (i.e., it will be always $|T_k| = 1$). A new test can *replace* the one in T_k only if it is *shorter* (which will depend on the problem domain, e.g. size measured in sequence of function calls in unit testing) or, if it is of the same size, then replace the current test only if the new test has better coverage on the other targets (i.e., sum of all the heuristic values on all targets).
- if the population is not full (i.e., $|T_k| < n$), then the test is added. Otherwise, if full (i.e., $|T_k| = n$), the test might replace the worst in the population, but only if not worse than it (but not necessarily better). This means no worse heuristic value or, if the same, no larger size.

The idea is that, for each target, we keep a population of candidate tests for it, for which we have at least some heuristic value. But once a target k is covered, we just need to store the best test, and discard the rest. Note that, if a discarded test in T_k was good for another target j, then it would be still stored in T_j anyway, so it is not lost.

When MIO needs to sample one test from the archive instead of generating one at random, it will do the following:

- choose one target k at random where $|T_k| > 0$ and k is not covered (i.e., no test has $h_k = 1$). If all non-empty populations are for covered targets, then just choose k randomly among them.
- choose one test randomly from T_k.

By using this approach, we aim at sampling tests that have non-zero heuristic (and so guidance) for targets that are not covered yet.

3.2. Exploration/Exploitation Control

In the MIO algorithm, the two main parameters for handling the trade-off between exploration and exploitation of the search landscape are the probability P_r of sampling at random and the population size n per target. Exploration is good at the beginning of the search, but, at the end, a more focused exploitation can bring better results. Like in Simulated Annealing, we use an approach in which we gradually reduce the amount of exploration during the search.

We define with F the percentage of time after which a focused search should start. This means that, for some parameters like P_r and n, we define two values: one for the start of the search (e.g., $P_r = 0.5$ and $n = 10$), and one for when the focused phase begins (i.e., $P_r = 0$ and $n = 1$). These values will linearly increase/decrease based on the passing of time. For example, if $F = 0.5$ (i.e., the focused search starts after 50% of the search budget is used), then after 30% of the search, the value P_r would decrease from 0.5 to 0.2.

Note, when during the search decreasing n leads to some cases with $|T| > n$, then those populations are shrunk by removing the worst individuals in it. Once the focused search begins (i.e., $P_r = 0$ and $n = 1$), then MIO starts to resemble a parallel (1+1) EA.

When dealing with many objectives, even if there is a clear gradient to cover them in the fitness landscape, there might be simply not enough time left to cover all of them. In software testing, the final user is only interested in tests that do cover targets, and not in tests that are heuristically close to cover them (e.g., close to solve complex constraints in some branch predicates, but not there yet). Therefore, between a test suite A that is close to but does not cover 100 targets, and another one B which does cover 1 target and is very far from covering the remaining 99, the final user would likely prefer B over A.

To take this insight into account, MIO tries to focus on just a few targets at a time, instead of spreading its resources thin among all the left uncovered targets. For example, in MIO there is an extra parameter m which controls how many mutations and fitness evaluations should be done on the same individual before sampling a new one. Like P_r and n, m varies over time, like starting from 1 and then increasing up to 10 when the focused search begins.

3.3. Feedback-Directed Sampling

When dealing with many objectives and limited resources, it might not be possible to cover all of them. As discussed in Section 3.2, the final user is only interested in the actually covered targets, and not on how close we are to cover them. Therefore, it makes sense to try to focus on targets that we have higher chances to cover. This is helpful also when dealing with infeasible targets for which any heuristic will just plateau at a certain point.

To handle these cases, we use a simple but yet effective technique that we call Feedback-Directed Sampling (FDS). The sampling algorithm from the archive discussed in Section 3.1 is modified as follow. Instead of choosing the target k randomly among the non-covered/non-empty ones, each of these targets will have a counter c_k. Every time a test is sampled from a population T_k, then c_k is increased by 1. Every time a new *better* individual is added to T_k (or replace one of its existing tests), then the counter c_k is reset to 0. When we sample from k from non-covered/non-empty ones, then, instead of choosing k at random, we choose the k with the lowest c_k.

As long as we get improvements for a target k, the higher chances will be that we sample from T_k, as c_k gets reset more often. On the other hand, for infeasible targets, their c will never be reset once they reach their plateau, and so they will be sampled less often. Similarly, more complex targets will be sampled less often, and so the search concentrates on the easier targets that are not covered yet. However, this is not an issue because, once an easy target k is covered, we do not sample from T_k any more (recall Section 3.1), unless also *all* the other targets are either covered or with empty T.

However, there is a potential issue with this kind of FDS based on *last* (most recent) improvement. For example, consider the case of d difficult, albeit feasible targets, for which it might take many fitness evaluations before getting an improvement. Their c counters will have same/similar values. However, instead of focusing in solving at least one of these feasible d targets within the given budget constraints, the FDS would spread resources equally among them. In other words, as soon as a difficult target j gets a c_j greater than the others, it will take at least $d-1$ fitness evaluations before sampling again from the population T_j. In those cases, it would make sense to concentrate resources to cover at least one of these d targets, instead of spreading such resources equally among these d targets and likely cover none of them.

A possible approach is to use a more *focused* FDS. For each target, we also keep track of how many steps s it took to get the last improvement in fitness for that target. In other words, when a target j gets a fitness improvement, we save $s_j = c_j$ before resetting $c_j = 0$. When choosing a T to sample from based on c counters, instead of choosing the target j based on the lowest c, we keep sampling from the previously sampled j, but only as long as $c \leq 2s$. In other words, once we get an improvement in fitness within s steps for a target j, we keep sampling from T_j for a number of times that is no more than twice it took to get to the previous improvement. Once the condition $c_j \leq 2s_j$ does not hold any longer for the current target j, we choose a different target to sample from with the *last* FDS approach.

Whether FDS improves or not performance is a matter of empirical investigation, as it depends on the proportion and type of infeasible targets in the SUTs. As such, FDS is not a core component of MIO, and rather a feature that can be (de)activated on demand.

4. Empirical Study

To evaluate the performance of the MIO algorithm, we compared it with random search, MOSA and WTS. We used two different case studies: (1) a set of artificial problems with varying, specific characteristics; (2) seven RESTful API web services.

In this paper, we aim at answering the following research questions:

RQ1: On which kinds of problem does MIO perform better than Random, MOSA and WTS?

RQ2: What is the impact of tuning parameters for exploration vs. exploitation of the search landscape in MIO and MOSA?

RQ3: How do the analysed algorithms fare on actual software?

4.1. Artificial Software

In this paper, we designed four different kinds of artificial problems. In all of them, there are z targets, and the search algorithm can be run for up to b fitness evaluations. A test is defined by two components: an *id* (e.g., think about it like the name of a method to call in unit testing) and a numeric integer value $x \in [0,r]$ (e.g., think about it like the input to a method call). Each target k is independent, and can be covered only by a test with $id = k$. The artificial problems will differ based on their fitness landscape. Given $g \in [0,r]$ the single global optimum chosen at random, and given the normalising function $\rho(d) = 1/(1+d)$ for distances, then we have four different cases for each target:

Gradient: $h_k = \rho(|x-g|)$. This represents the simplest case where the search algorithm has a direct gradient from x toward the global optimum g.

Plateau: $h_k = \rho(g-x)$ if $g \geq x$, else $h_k = \rho(0.1 \times r)$. In this case, we have one side of the search landscape (before the value of the global optimum g) with a clear gradient. However, the other side is a plateau with a relatively good fitness value (note that $0 \leq |g-x| \leq r$).

Deceptive: $h_k = \rho(g-x)$ if $g \geq x$, else $h_k = \rho(1+r-x)$. This is similar to the *Plateau* case, where one side of the search landscape has a clear gradient toward g. However, the other side has a deceptive gradient toward leaving g and reach the maximum value r.

Infeasible: like *Gradient*, but with a certain number of the z targets having a constant $h_k = \rho(1)$ and no global optimum.

We implemented the four search algorithms in which, when a test is sampled, its *id* and x values are chosen at random within the given valid ranges. Mutations on x is done by adding/subtracting 2^i, where i is chosen randomly in [0,10]. We consider mutating *id* as a *disruptive* operation, and, as such, we only mutate it with low probability 0.01. Mutating *id* means changing both *id* and x at random (think about mutating a function call with string inputs into another one that requires integers, where the strings x would have no meaning as integers). All the analysed search algorithms use the same random sampling, mutation operation and archive to store the best tests found so far.

For the MIO algorithm, we used $F = 0.5$, $P_r = 0.5$, $n = 10$ and max mutations 10. For MOSA, we used the same settings as in [7], i.e. population

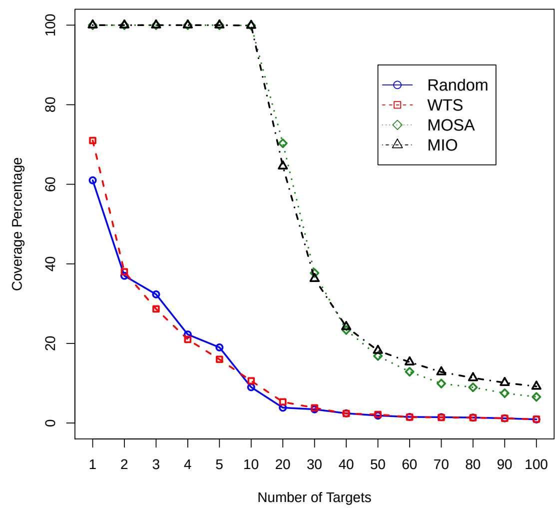


Figure 3: Coverage results on the *Gradient* problem type, with varying number of targets z.

size 50 and tournament selection size 10. WTS uses the same population size as MOSA, with up to 50 test cases in the same test suite (i.e., one individual). A randomly sampled test suite in WTS will have size randomly chosen between 1 and 50. WTS also has mutation operators to add a new test (probability 1/3) in a test suite, remove one test at random (probability 1/3), or modify one (probability 1/3) like in MIO and MOSA. WTS also uses a crossover operator with probability 70% to combine test suites.

For each problem type but *Infeasible*, we created problems having a variable number of z targets, in particular $z \in \{1,2,3,4,5,10,20,30,40,50,60,70,80,90,100\}$, i.e., 15 different values in total, ranging from 1 to 100. We used $r = 1000$. We ran each of the four search algorithms 100 times with budget $b = 1000$. As the optima g are randomised, we make sure that the search algorithms run on the same problem instances. In the case of the *Infeasible* type, we used 10 *Gradient* targets, on which we added a different number of infeasible targets in $\{0,1,2,3,4,5,10,20,30,40,50,60,70,80,90,100\}$, i.e., 16 values in total, with z ranging from $(10 + 0) = 10$ to $(10 + 100) = 110$. Figure 3 shows the results for the *Gradient* type, Figure 4 for *Plateau*, Figure 5 for *Deceptive*, and Figure 6 for *Infeasible*.

The *Gradient* case (Figure 3) is the simplest, where the four search

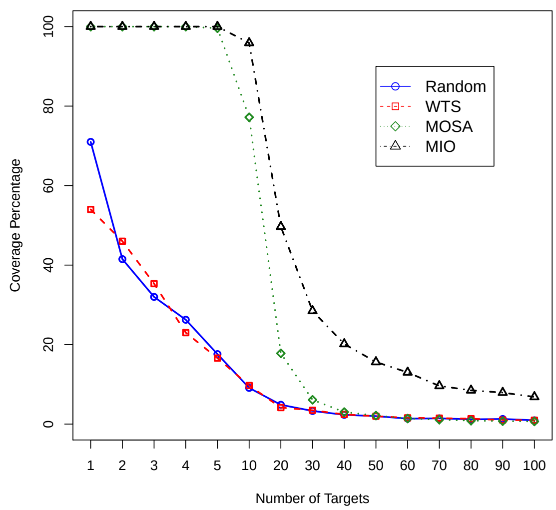


Figure 4: Coverage results on the *Plateau* problem type, with varying number of targets z.

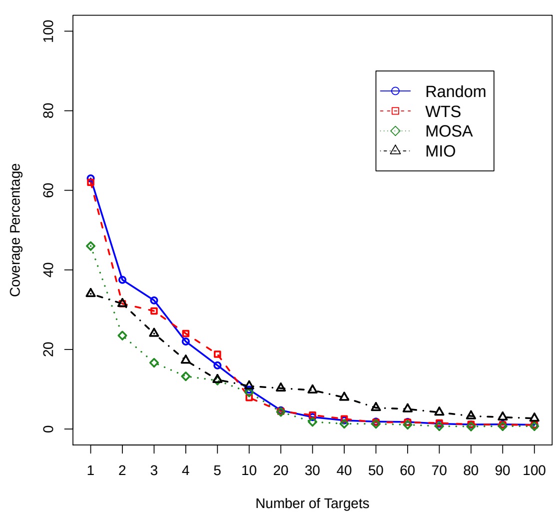


Figure 5: Coverage results on the *Deceptive* problem type, with varying number of targets z.

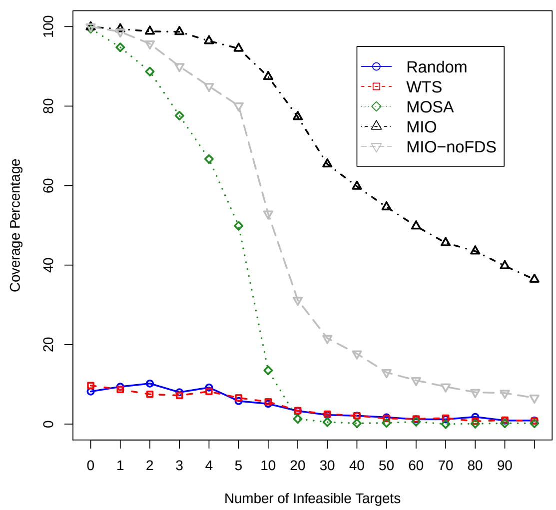


Figure 6: Coverage results on the *Infeasible* problem type, with varying number of infeasible targets on top of 10 *Gradient* ones.

algorithms obtain their highest coverage. MIO and MOSA have very similar performance, which is higher than the one of Random and WTS. However, on the more difficult case of *Plateau* (Figure 4), MIO starts to have a clear advantage over MOSA. For example, from $z = 30$ on, MOSA becomes equivalent to Random and WTS, covering nearly no target. However, in that particular case, MIO can still achieve around 20% coverage (i.e., 6 targets). Even for large numbers of targets (i.e., 100 when taking into account that the search budget b is only 1000), still MIO can cover some targets, whereas the other algorithms do not.

The *Deceptive* case (Figure 5) is of particular interest: for low numbers of z targets (i.e., up to 10), both MIO and MOSA perform worse than Random. From 10 targets on, MOSA is equivalent to Random and WTS, whereas MIO has better results. This can be explained by taking into account two contrasting factors: (1) the more emphasis of MIO and MOSA on exploitation compared to the exploration of the search landscape is not beneficial in deceptive landscape areas, whereas a random search would not be affected by it; (2) MIO does better handle large numbers of targets (Figure 3), even when there is no gradient (Figure 4). The value $z = 10$ seems to be the turning point where (2) starts to have more weight than (1).

The *Infeasible* case (Figure 6) is where MIO obtains the best results compared to the other algorithms. For this case, we also ran a further version of MIO in which we deactivated FDS (recall Section 3.3), as we wanted to study its impact in the presence of infeasible targets. From 20 infeasible targets on, MOSA, Random and WTS become equivalent, covering nearly no target. However, MIO can still cover nearly 80% of the 10 feasible testing targets. For very large numbers of infeasible targets like 100, still MIO can cover nearly 40% of the feasible ones. This much better performance is mainly due to the use of FDS (see the gap in Figure 6 between MIO and MIO-noFDS). However, even without FDS, MIO still does achieve better results compared to the other algorithms.

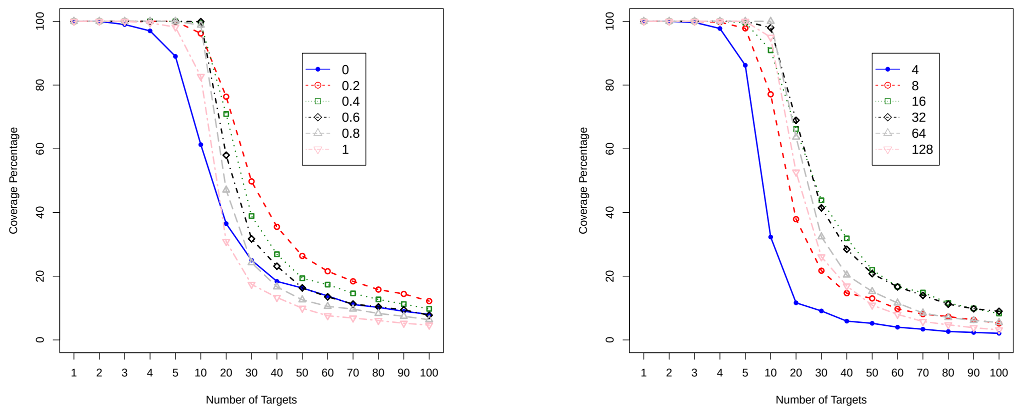
> **RQ1**: *on all the considered problems, MIO is the algorithm that scaled best. Coverage improvements were even up to 80% in some cases.*

When using a search algorithm, some parameters need to be set, like the population size or crossover probability in a GA. Usually, common settings in the literature can already achieve good results on average [18]. Finding tuned settings that work better on average on a large number of different artefacts is not trivial. Ideally, a user should just choose for how long a search algorithm should run, and not do long tuning phases by himself on his problem artefacts. Parameter tuning can also play a role in algorithm comparisons: what if a compared algorithm performed worse just because one of its chosen settings was sub-optimal?

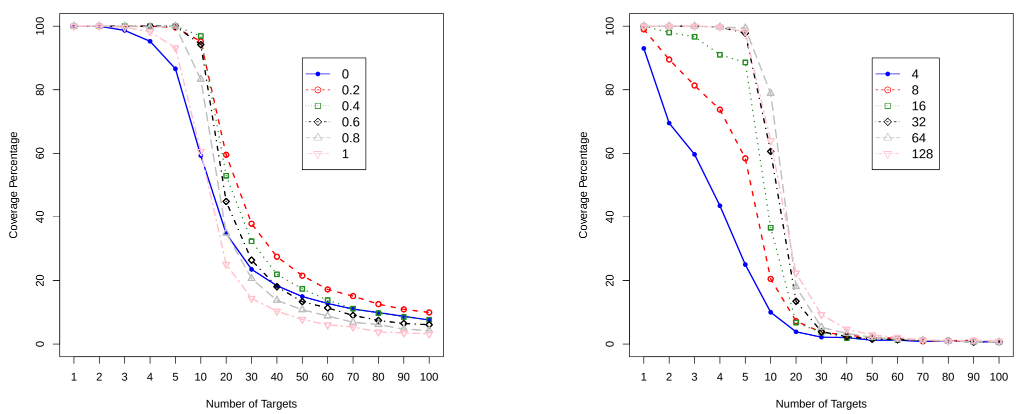
Arguably, among the most important parameters for a search algorithm are the ones that most impact the tradeoff between the exploration and the exploitation of the search landscape. In the case of MIO, this is clearly controlled by the F parameter (low values put more emphasis on exploitation, whereas for high values a large number of tests are simply sampled at random). In the case of population-based algorithms, the population size can be considered as a parameter to control such tradeoff. Small populations would reward exploitation, whereas large populations would reward exploration.

To study these effects, we carried out a further series of experiments on the *Gradient*, *Plateau* and *Deceptive* problem types. For MIO, we studied six different values for F, in particular $\{0,0.2,0.4,0.6,0.8,1\}$. For MOSA, we studied six different values for the population size, i.e. $\{4,8,16,32,64,128\}$. Each experiment was repeated 100 times. Figure 7 shows the results of these experiments.

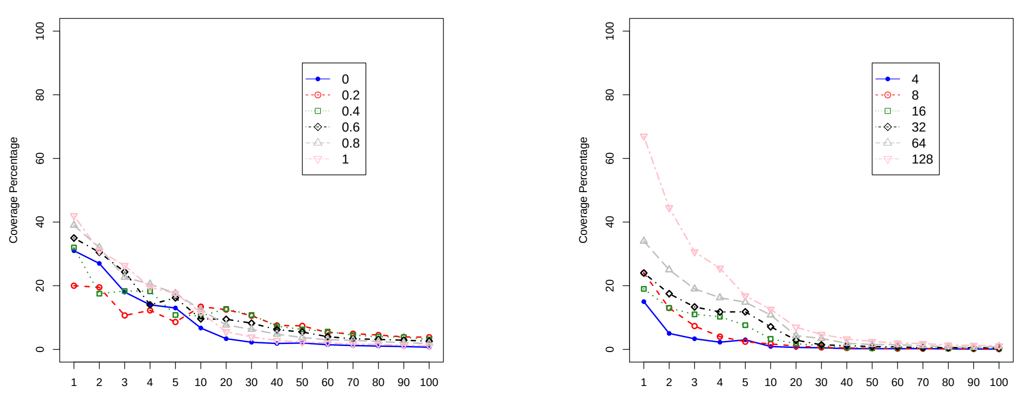
For MIO, the results in Figure 7 do match expectation: for problems



(a) *Gradient* problem type.



(b) *Plateau* problem type.



(c) *Deceptive* problem type.

Figure 7: Tuning of F for MIO (left side) and population size for MOSA (right side).

with clear gradient or with just some plateaus, a more focused search that rewards exploitation is better. The best setting is a low $F = 0.2$, although the lowest $F = 0$ is not particularly good. You still need some genetic diversity at the beginning of the search, and not rely on just one single individual. For deceptive landscapes, exploration can be better, especially for a low number of targets. For example, with $z = 1$ then $F = 1$ provides the best performance. However, for larger number of targets, too much exploration would not be so beneficial, as it would not have enough time to converge to cover the targets.

In the case of MOSA, Figure 7 provides some interesting insight. For simple problems with clear gradient, one would expect that a focused search should provide better results. However, the small population size of 4 is actually the configuration that gave the worst results. The reason is that there is only little genetic material at the beginning of the search, and new one is only generated with the mutation operator. However, a too large population size would still be detrimental, as not focused enough. In that particular problem type, the best population size seems to be ranging from 16 to 32, i.e., not too large, but not too small either. In case of plateaus, still a too small population size (e.g., 4) gives the worst result. However, in case of plateaus, there is a need to have some more exploration in the search landscape, and this is confirmed by the fact that the best results are obtained with large population sizes (e.g., 64 and 128). This effect is much more marked in the case of deceptive landscapes, where large population sizes lead to much better results.

The experiments reported in Figure 7 clearly points out to a challenge in population-based algorithms when dealing with many-objective problems. A too small population size would reduce diversity in the initial genetic material. But a too large population size would hamper convergence speed. Finding a fixed, right population size that works on most problem sizes (e.g., $z = 10$ vs $z = 1{,}000{,}000$) might not be feasible. To overcome this issue, MIO uses a dynamically sized population, whereas the tradeoff between exploration and exploitation is controlled by a dynamically decreasing probability P_r of creating new tests at random (instead of mutating the current ones stored in the archive).

RQ2: *On the analysed problems, the population size and the F parameter have clear effects on performance, which strongly depend on whether on the given problem one needs more or less exploitation/exploration.*

Table 1: Information about the seven RESTful web services used in the empirical study. We report their number of Java/Kotlin classes and lines of code. We also specify the number of endpoints, i.e., the number of exposed resources and HTTP methods applicable on them, and also whether these SUTs interact with a database.

Name	# Classes	LOCs	Endpoints	Database
catwatch	69	5442	23	yes
features-service	23	1247	18	yes
proxyprint	68	7534	115	yes
rest-ncs	9	602	6	no
rest-news	10	715	14	yes
rest-scs	13	859	11	no
scout-api	75	7479	49	yes
Total	267	23878	236	5/7 yes

4.2. Real Software

When designing algorithms to work on a large class of problems, it is common to evaluate them on artificial problems to try to abstract away and analyse in details the characteristics for which such algorithms perform best. For example, the very popular NSGA-II algorithm (on which MOSA is based on) was originally evaluated only on nine numerical functions [15]. However, using only artificial problems is risky, as those might abstract away some very important factors. A good example of this issue is Adaptive Random Testing, where artificial problems with artificially high fault rates were masking away its prohibitive computational cost [19].

To mitigate this issue, we also carried out experiments on actual software, using the EvoMaster tool, where we implemented the MIO algorithm besides WTS and MOSA. EvoMaster generates system level test cases for RESTful API web services. Seven APIs were chosen for the experiments, and they are described in Table 1. To be able to replicate the experiments in this paper, we provide this case study online on GitHub[7].

The APIs *features-service*, *proxyprint*, *scout-api* and *catwatch* are actual open-source Java projects selected from GitHub, using different technologies like Spring and Dropwizard. These four APIs use a SQL database. On

[7]https://github.com/EMResearch/EMB

the other hand, *rest-news* is an artificial SpringBoot application written in Kotlin, which has been used in didactic settings. Similarly, *rest-ncs* (REST Numerical Case Study) and *rest-scs* (REST String Case Study) are artificial APIs using SpringBoot and that contain code previously used for experiments in unit testing of classes heavily dependent on numerical [19] and string [20] computations.

We ran MIO, MOSA, WTS and random search on each of the seven web services. For MIO, we considered three variants: without FDS, *last* FDS and *focused* FDS. Each algorithm was run with the same search budget of 100 thousand HTTP calls. Each experiment was repeated 50 times with different random seeds. Each experiments required at least three CPUs, as three processes needed to be run in parallel and the SUTs are multi-threaded. Therefore, a large cluster of computers was required to run all these experiments, which needed a total of 632 days of CPU time.

When comparing algorithms, it is also important to study their computational cost. An algorithm could be more efficient, and run more fitness evaluations given the same amount of time. But the computational cost is strongly affected by the low level details of the code implementation, and so it poses further threats to validity for fair comparisons. We did not consider the computational cost in this paper because, for system testing, the cost of a fitness evaluation (marshalling of JSON data, HTTP over TCP and access to SQL databases) is much higher than any overhead of the used algorithm. This, however, might not be the case in the context on unit testing, where using time as stopping criterion would be more appropriate (as such algorithm overheads might not be negligible compared to the cost of the fitness function).

Average values of these experiments are reported in Table 2, where we also report the Vargha-Delaney effect sizes $\hat{A}_{12}$ and the results of Mann-Whitney-Wilcoxon U-tests at $\alpha = 0.05$ level [21]. Table 3 reports the relative rank in performance among the different algorithms, which is statistically significant according to the Friedman test. Note that, for MIO, in these tables we only report the *focused* FDS variant.

From what can be seen in Table 2 and Table 3, the MIO algorithm has overall the best results. However, it is not always the best, i.e., only on three out of seven SUTs. On the other hand, on these other SUTs MIO is the second best algorithm. Random search is generally the worst algorithm. MOSA and WTS have inconsistent results. For example, although MOSA is the best on *rest-ncs* and *rest-scs*, it is also the worst on *proxyprint*, even

Table 2: Comparions of algorithms on the seven web services. Coverage is not a percentage, but rather the average raw sum of covered targets used in the fitness function of EvoMaster. For each algorithm, we also specify if better than any of the others, i.e. $\hat{A}_{12} > 0.5$ (in parenthesis) and p-value less than 0.05.

SUT	Algorithm	Tests	Coverage	Better than
catwatch	MIO	51.2	1000.0	MOSA(0.64) RAND(0.81) WTS(0.75)
	MOSA	52.3	991.3	RAND(0.72) WTS(0.63)
	RAND	52.5	985.6	
	WTS	49.9	990.6	RAND(0.64)
features-service	MIO	51.6	558.8	RAND(0.78)
	MOSA	52.2	551.4	RAND(0.76)
	RAND	38.1	479.1	
	WTS	63.0	592.4	MIO(0.66) MOSA(0.69) RAND(0.87)
proxyprint	MIO	258.0	1559.8	MOSA(0.95) RAND(0.85) WTS(0.84)
	MOSA	298.3	1526.5	
	RAND	300.4	1534.1	MOSA(0.63)
	WTS	294.5	1538.9	MOSA(0.74)
rest-ncs	MIO	55.2	525.6	RAND(1.00) WTS(0.86)
	MOSA	51.9	526.5	RAND(1.00) WTS(0.90)
	RAND	41.9	436.4	
	WTS	48.4	515.9	RAND(1.00)
rest-news	MIO	30.3	265.0	
	MOSA	31.1	265.3	RAND(0.63)
	RAND	31.1	262.7	
	WTS	31.8	265.3	
rest-scs	MIO	73.5	594.1	RAND(1.00) WTS(0.98)
	MOSA	109.6	722.0	MIO(1.00) RAND(1.00) WTS(1.00)
	RAND	34.3	510.2	
	WTS	53.1	534.4	RAND(0.99)
scout-api	MIO	173.2	1829.5	RAND(1.00)
	MOSA	188.4	1828.0	RAND(1.00)
	RAND	177.1	1427.8	
	WTS	200.9	1944.0	MIO(0.99) MOSA(0.99) RAND(1.00)

worse than random search. Similarly, WTS is the best on *features-service* and *scout-api*, but then it is the second-worst on four of the remaining SUTs.

To shed more light on these different performance behaviours, Figure 8 plots the average number of covered targets through time. The behaviour

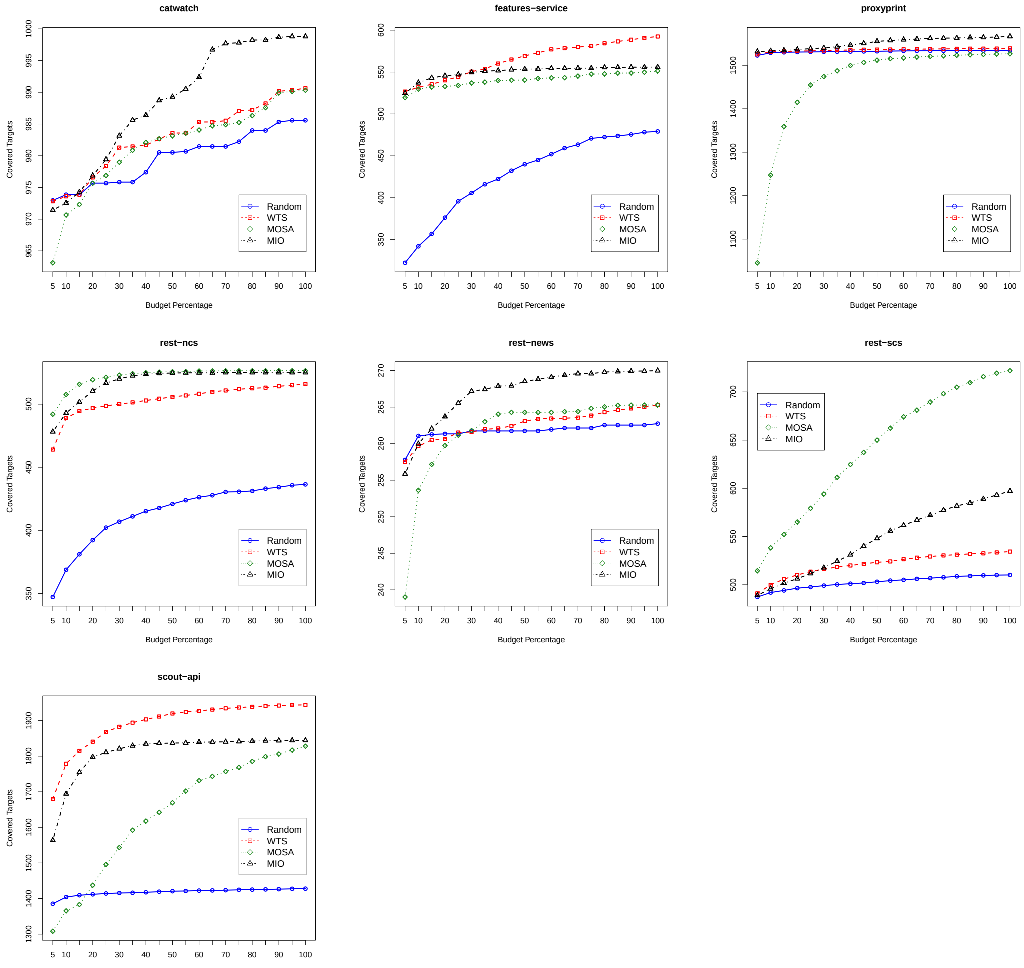


Figure 8: Performance over time of the different search algorithms.

Table 3: Algorithm ranks based on their average performance, where Rank 1 represents the highest achieved coverage. We also report the χ^2 and p-value of the Friedman test.

SUT	MIO	MOSA	WTS	RAND
catwatch	1	2	3	4
features-service	2	3	1	4
proxyprint	1	4	2	3
rest-ncs	2	1	3	4
rest-news	1	2	3	4
rest-scs	2	1	3	4
scout-api	2	3	1	4
Average	1.6	2.3	2.3	3.9
χ^2=11.74286, p-value=0.008317995				

of MOSA is of particular interest. During its initial phase, it is worse than random search on *catwatch*, *rest-news* and *scout-api*. It takes it 20%-25% of the search budget before becoming better than random search. However, on *rest-ncs* and *rest-scs* it has a clear better margin already from the very beginning, i.e. in the first 5% of the budget. On *rest-scs* in particular, MOSA has a very large margin over all the other algorithms.

At the current moment, we are unable to explain such peculiar behaviour. But, to design better algorithms, it will be important to study in detail why this kind of performance improvements are obtained. However, to enable researchers to get such kind of insight, it will be necessary to develop tools to visualize the evolution of these algorithms through time. For example, if such kind of behaviour is linked to some specific properties of the SUTs, and if such properties can be detected before starting a search, a hybrid approach could be designed to choose the best algorithm (e.g., MIO or MOSA) to employ based on those detected characteristics.

To address the potential issue of infeasible targets, the MIO algorithm employs the FDS technique, with two different variants: *last* and *focused*. Comparisons of these variants of FDS are shown in Table 4. On artificial software, FDS gives very large benefits (e.g., recall Figure 6). However, on actual software, improvements are relatively modest, if any at all. In two cases (*proxyprint* and *rest-news*), *focused* FDS gives statistically better results. However, on two other cases (*rest-scs* and *scout-api*), FDS actually

Table 4: Comparions of MIO FDS variants on the seven web services. Coverage is not a percentage, but rather the average raw sum of covered targets used in the fitness function of EvoMaster. For each variant, we also specify if better than any of the others, i.e. $\hat{A}_{12} > 0.5$ (in parenthesis) and p-value less than 0.05.

SUT	FDS	Tests	Coverage	Better than
catwatch	NONE	51.4	1003.7	
	LAST	51.2	1000.0	
	FOCUSED	51.3	998.8	
features-service	NONE	51.5	552.4	
	LAST	51.6	558.8	
	FOCUSED	50.8	555.9	
proxyprint	NONE	268.5	1546.6	
	LAST	258.0	1559.8	NONE(0.70)
	FOCUSED	261.2	1566.3	NONE(0.73)
rest-ncs	NONE	55.3	526.5	
	LAST	55.2	525.6	
	FOCUSED	54.8	525.1	
rest-news	NONE	30.0	267.2	
	LAST	30.3	265.0	
	FOCUSED	30.5	270.0	LAST(0.62)
rest-scs	NONE	79.6	609.7	LAST(0.62)
	LAST	73.5	594.1	
	FOCUSED	73.3	597.2	
scout-api	NONE	177.7	1883.0	LAST(0.71) FOCUSED(0.69)
	LAST	173.2	1829.5	
	FOCUSED	174.7	1844.3	

gives statistically *worse* results.

A possible explanation is that, in the artificial examples, all targets were completely independent. However, on actual software, there can be dependencies. For example, a test case whose execution does reach an infeasible target can also reach other targets. Consider this simple example:

```
void foo(int x, int y){
   if(complexPredicate(x)){
```

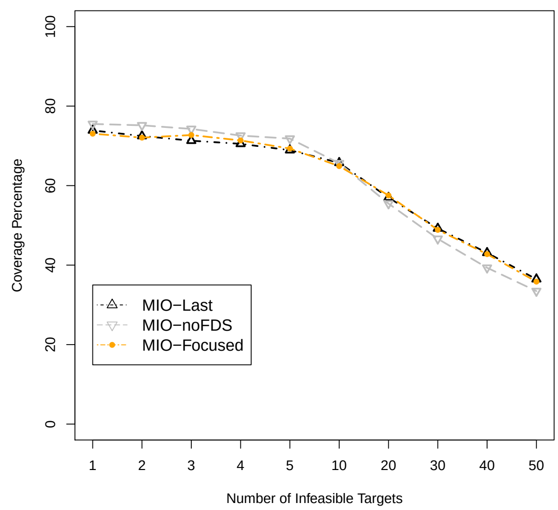


Figure 9: Coverage results in the example of high reachability among targets, with increasing number of infeasible targets out of 100.

```
      if(infeasiblePredicate(y)){
        // target i
      }
      if(feasiblePredicate(y)){
        // target j
      }
    }
}
```

The target i is infeasible. However, any test that does reach the predicate for i does also necessarily reach the predicate for target j. Therefore, every time we sample from the population for the infeasible i we still have tests that could be good for other targets as well. In this particular example, sampling from i would still give you the right input x value to reach target j. However, this does not fully explain why in some cases the use of FDS does reduce performance.

To provide more insight, we carried out an experiment to study how FDS would behave in such type of context. We created an artificial example to represent maximum dependency among targets reachability (each target is always reached, albeit not necessarily covered). In this case, optimised tests for infeasible targets are still useful, because they always reach as well all the other feasible targets that are not covered yet.

We considered a function with two inputs, x and y. There are 100 testing targets, 50 in the form `if(x == ?)`, and 50 in the form `if(y == ?)`. These branches are all at the same level (no nesting), and contain no return statements. Therefore, when an input pair x,y is given as input, each single target is reached, and its predicate is evaluated.

The constants in the `if` statements are sampled between 0 and 10,000. The algorithms do not generate values out of this range. We considered cases in which we inject a certain number of infeasible targets for the x branches, in which the constant is -1 (the target is so not reachable, and the algorithms have gradient to decrease x down till 0). In particular, we use the following number of infeasible targets: $\{1,2,3,4,5,10,20,30,40,50\}$. We ran MIO in three modes: without FDS, and with *last* and *focused* FDS modes. Search budget was 10,000 fitness evaluations. Experiments were repeated 100 times.

Figure 9 shows the results of this experiment. On one hand, for low numbers of infeasible targets, FDS gives slightly worse results. On the other hand, for high numbers of infeasible targets, FDS gives better results.

This example is deliberately constructed to represent a case in which FDS would not be expected to be particularly useful, even in the presence of a large proportion of infeasible targets. So, it is not unexpected that, even with 50% of infeasible targets, FDS only provides minor improvement. However, for small numbers of infeasible targets (up to 10%), FDS actually gives slightly worse results, similarly to some of the experiments on actual software. It seems like that the use of FDS does have impact on the tradeoff between exploration and exploitation of the search landscape. Depending on the search landscape, this can improve or worsen the performance. For small number of infeasible targets, the impact of such difference seems larger than the benefits of handling the infeasible targets. However, detailed control flow analyses of these cases and landscape analyses will be required to shed more light on this kind of dynamics.

RQ3: *the experiments on actual software are consistent with the ones on artificial problems: the MIO algorithm still achieves the best results, but not on all problems.*

5. Threats to Validity

Threats to *internal* validity come from the fact that our study is based on the EvoMaster tool. We cannot guarantee that it is bug free. However, to reduce such threat, not only EvoMaster has been carefully tested, but also its source code is available for review as an open-source project hosted on GitHub.

It is possible that, when comparing a new algorithm with the existing state-of-the-art, better performance might be just due to wrong implementation of such existing algorithms due to misunderstanding of their details. In the case of WTS, we were among its original co-authors [6] . In the case of MOSA, our implementation was reviewed and fixed by one of its co-authors[8].

Randomized algorithms are affected by chance. To keep this factor under control, each experiment was repeated either 50 or 100 times. The appropriate statistical tests and effect sizes were then employed to analyze the results.

Threats to *external* validity come from the fact that only seven web services for a total of 23 thousand lines of code were used for the empirical evaluation on real software. This was due to the large cost of experiments on system level test generation, which required more than 600 days of computational effort. Furthermore, albeit very common in industry, RESTful APIs are not so common among open-source projects. The selected sample is biased in regard of what could be found and compiled/run without problems.

6. Conclusion

In this paper, we have presented a novel search algorithm that is tailored for the problem of generating test suites, in particular for system testing. We call it the Many Independent Objective (MIO) algorithm. We have carried out an empirical study to compare MIO with the other main algorithms for test suite generation: the Whole Test Suite (WTS) approach and the Many-Objective Sorting Algorithm (MOSA). We also used random search as a baseline.

On artificial problems with increasing complexity, MIO achieved better results than the other algorithms. Such improvements were also confirmed when using MIO for test suite generation for system testing of seven RESTful API web services. However, MIO did not give the best result on all of these web services.

Future work will focus on analyzing and understanding the reasons why, in some cases, MOSA and WTS gave better results than MIO. This will be

[8]https://github.com/EMResearch/EvoMaster/pull/1

essential to design novel MIO variants to improve performance even further. However, system level test suite generation is a very complex task to analyze and understand. Therefore, one essential task will be to implement tools to visualize the evolution of test suites throughout the search, with the aim of helping researchers to analyze those complex dynamics.

To help researchers integrate MIO in their frameworks, all the code used for the experiments in this paper is available online on a public repository, as part of the EvoMaster tool at `www.evomaster.org`.

Acknowledgments. We would like to thank Annibale Panichella for his help with the implementation of the MOSA algorithm. This work is supported by the Research Council of Norway (project on Evolutionary Enterprise Testing, 274385), and by the National Research Fund, Luxembourg (FNR/P10/03).

[1] M. Harman, S. A. Mansouri, Y. Zhang, Search-based software engineering: Trends, techniques and applications, ACM Computing Surveys (CSUR) 45 (1) (2012) 11.

[2] D. H. Wolpert, W. G. Macready, No free lunch theorems for optimization, IEEE Transactions on Evolutionary Computation 1 (1) (1997) 67--82.

[3] S. Ali, M. Z. Iqbal, A. Arcuri, L. C. Briand, Generating test data from OCL constraints with search techniques, IEEE Transactions on Software Engineering (TSE) 39 (10) (2013) 1376--1402.

[4] G. Fraser, A. Arcuri, Achieving scalable mutation-based generation of whole test suites, Empirical Software Engineering (EMSE) 20 (3) (2015) 783--812.

[5] B. Li, J. Li, K. Tang, X. Yao, Many-objective evolutionary algorithms: A survey, ACM Computing Surveys (CSUR) 48 (1) (2015) 13.

[6] G. Fraser, A. Arcuri, Whole test suite generation, IEEE Transactions on Software Engineering 39 (2) (2013) 276--291.

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

[8] A. Arcuri, Many Independent Objective (MIO) Algorithm for Test Suite Generation, in: International Symposium on Search Based Software Engineering (SSBSE), 2017, pp. 3--17.

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

[10] S. Ali, L. Briand, H. Hemmati, R. Panesar-Walawege, A systematic review of the application and empirical investigation of search-based test-case generation, IEEE Transactions on Software Engineering (TSE) 36 (6) (2010) 742--762.

[11] G. Fraser, A. Arcuri, EvoSuite: automatic test suite generation for object-oriented software, in: ACM Symposium on the Foundations of Software Engineering (FSE), 2011, pp. 416--419.

[12] G. Fraser, A. Arcuri, A large-scale evaluation of automated unit test generation using EvoSuite, ACM Transactions on Software Engineering and Methodology (TOSEM) 24 (2) (2014) 8.

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

[14] J. M. Rojas, M. Vivanti, A. Arcuri, G. Fraser, A detailed investigation of the effectiveness of whole test suite generation, Empirical Software Engineering (EMSE) (2016) 1--42.

[15] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Transactions on Evolutionary Computation (TEVC) 6 (2) (2002) 182--197.

[16] A. Arcuri, RESTful API Automated Test Case Generation, in: IEEE International Conference on Software Quality, Reliability and Security (QRS), IEEE, 2017, pp. 9--20.

[17] P. McMinn, Search-based software test data generation: A survey, Software Testing, Verification and Reliability 14 (2) (2004) 105--156.

[18] A. Arcuri, G. Fraser, Parameter tuning or default values? an empirical investigation in search-based software engineering, Empirical Software Engineering (EMSE) 18 (3) (2013) 594--623.

[19] A. Arcuri, L. Briand, Adaptive random testing: An illusion of effectiveness?, in: ACM Int. Symposium on Software Testing and Analysis (ISSTA), 2011, pp. 265--275.

[20] M. Alshraideh, L. Bottaci, Search-based software test data generation for string data using program-specific search operators, Software Testing, Verification, and Reliability 16 (3) (2006) 175--203. doi:http://dx.doi.org/10.1002/stvr.v16:3.

[21] A. Arcuri, L. Briand, A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering, Software Testing, Verification and Reliability (STVR) 24 (3) (2014) 219--250.