Automatic Benchmark Generation

Adrien Mathieu Guillermo Polito*

Benchmarking and testing share strong similarities in the techniques used, yet, to the best of our knowledge fuzzing, hasn't been use to automatically detect performance issues, despite its proficiency in automatic bug detection.

We show that these techniques can indeed be adapted to automatically generate interesting benchmark case, and, in doing so, we explore several fuzzing variants. We also show how to take advantage of pure object-oriented aspects of programming languages to solve common fuzzing pitfalls.

Introduction

Unit testing and benchmarking serve a similar purpose: to detect a regression of capabilities of a given software. Indeed, they even share most of the techniques used (such as coverage evaluation to assert the exhaustivity of the tests and benchmarks), and are commonly ran together in the CI. However, testing techniques are not limited to unit testing. For instance, a popular family of testing techniques, called fuzz testing [1], consists in automatically generating inputs to actively try to find bugs [2], rather than manually developing test cases. This allows discovering bugs that the developer has not thought of while writing test cases. Yet, to the best of our knowledge, nobody has tried fuzzing benchmark cases to actively seek for performance issues.

To this end, we explore how fuzzing techniques can be applied to automatic benchmark generation. We find that input generation techniques similar to the one used for test generation work reliably for our purposes too. However, we also find that there is a significant difference in the detection of a performance bug, compared to a behavioral bug. This is, in part, due to the very poor performance specifications given with a program, not to say that there are usually none.

Besides the core fuzzing techniques, modern fuzzers collect feedback information during an instrumented execution of the tested program. This allows them to reason about the causality between the characteristics of the data fed into the program, and its behavior. For performance fuzzing, measuring the execution time is not enough to do the same, because it is significantly affected by a lot of factors outside of the input of a program. Instead we measure a more consistent proxy, the number of message sends, by instrumenting the Pharo virtual machine (VM) itself.

*Internship supervisor

Our contributions. To automatically generate inputs that exhibit poor performance behavior, we have developed an automatic benchmark generation framework, `Gnocco`, within the Pharo system. Given a program (or several programs, for a comparative benchmark), and a grammar specifying its valid inputs (or a subset), it will setup the required instrumentation, then mutate the grammars until they generate interesting cases. The mutation is performed under the guidance of one of the available metaheuristics provided with the framework, which use the feedback collected during the execution of the program on the generated inputs.

The implementation is modular: any metaheuristic can be plugged in to guide generation, so the user can easily fine-tune a provided metaheuristic, or even use a new one. Besides, the guiding system is agnostic on the underlying metric. The one provided can easily be replaced to make it match the definition of performance a user might want to use.

1. Motivations

Let's identify the key difficulties of automatic benchmark generation with an example, in Pharo, a pure object-oriented programming language heavily inspired by Smalltalk.

Pharo has several serializers in its standard library, among which are STON [3] and Fuel[1]. Fuel's aim is to provide fast serialization and deserialization targeting a compact, binary format. On the other hand, STON targets a superset of JSON which can encode arbitrary typed object graphs. Their performance is expected to be linear, as can be seen in Figure 1. Due to the different provided feature set, Fuel is usually faster than STON.

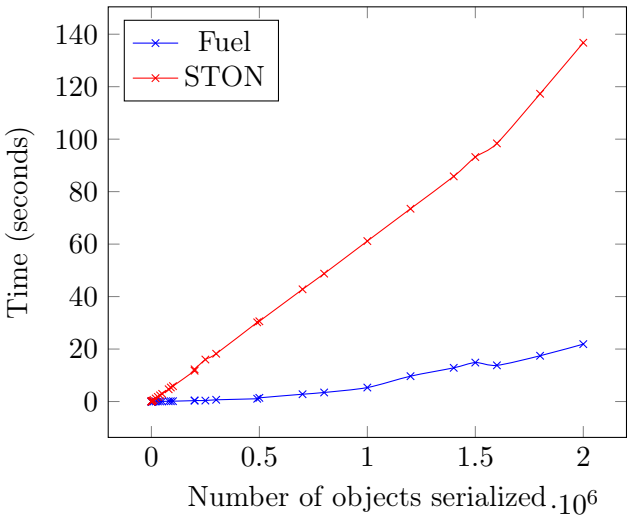

Figure 1: Benchmarks comparing STON and Fuel performance

Yet, an issue[2] was reported by some users witnessing an important serialization speed decrease when the size of the data they were serialization was over a certain threshold. The results of a benchmark exhibiting that issue are presented in Figure 2.

This issue is due to the fact that Fuel keeps instances of a common class that it has already handled in a hash map, using the pointer equality for comparison, since two structurally equal objects need to be serialized separately. The problem is that, since

[1] `https://theseion.github.io/Fuel/`
[2] `https://github.com/theseion/Fuel/issues/269`

Pharo's garbage collector can freely move objects around at any time, the pointer itself can only be used for equality, and not for hashing. To circumvent this, Pharo stores 22 bits of hash in each object's header, which are used in place of the pointer value for hashing. This means that the hash space is limited to $2^{22} \approx 4 \cdot 10^6$ values. Beyond that number of objects, collisions become more and more common, making each access Fuel does to its hash map very expensive.

This example shows that performance bugs can be challenging to identify (the issue with identity hash maps has gone unnoticed for several years in Pharo, despite identity hash maps being widely used). This is in part due, like other bugs, to the specificity of the conditions that need to be fulfilled to exhibit the bug. Since, for behavioral bugs, fuzzing has been successfully applied for automatically creating examples that meet conditions that trigger bugs, we naturally investigate whether the same techniques can be used to find performance bugs.

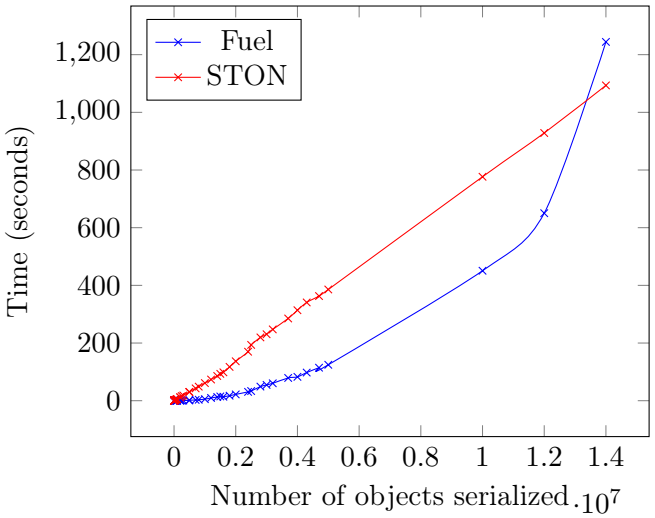

Figure 2: Benchmarks illustrating the performance issue with a large number of similar objects

Any improvement in the tools used to assist developers debugging a performance issue can be very much impactful, as these bugs require more time to be solved than other kind of bugs, and are usually assigned more experienced programmers, showing that they are hard to fix [4].

2. Fuzzing benchmark cases

Fuzz testing is a testing technique that involves generating or mutating randomly inputs to trigger unusual execution paths in a given program [1] [5] [6] [7]. To improve the efficiency of fuzzing, input generation techniques have been improved, avoiding generating inputs that are sufficiently invalid to make the program stop almost immediately. For instance, generative grammars can be used to ensure the validity of the generated input, in order to go further in the program execution than just its input validation step.

2.1. Generative grammar

Among the various techniques used by fuzzers to generate interesting input, we need one that guarantees the validity of the generated input; otherwise, we waste computational resources by having a generated input rejected by the validation step of the benchmarked program.

To this end, we use generative grammars. The user defines a grammar which generates a language of valid inputs (or picks among the provided grammars). The user also provides some constraints about the size of the generated input. Then, our framework uses that grammar to generate valid inputs.

To tweak the generation, and guide it towards interesting cases, each rule of the grammar has an associated weight. When choosing a rule to expand a non-terminal, our framework will take into account the relative weight of all the rules that can be applied, taking into account the constraints. This allows a fine control about how much each rule will be used, on average.

For instance, if we wanted to generate valid French mobile phone numbers, we would do as follow

```
1  defineGrammar
2
3    ntPhoneNumber -->
4      ntStartBlock, ' ', ntBlock, ' ', ntBlock, ' ', ntBlock, ' ', ntBlock.
5    ntStartBlock --> '06' | '07'.
6    ntBlock --> ($0 - $9), ($0 - $9).
7
8    ^ ntPhoneNumber
```

which defines a grammar with three non-terminals, `ntPhoneNumber`, `ntStartBlock` and `ntBlock`. `ntPhoneNumber` is marked as the starting non-terminal on the last line.

2.2. Graph generation

In pure object-oriented languages, data are typed object graphs. Instead of generating input as strings, we could directly generate graphs of objects. The advantage is that strings only work if the benchmarked program has a parser that converts string inputs to their internal data representation. While this is usually the case for applications, libraries usually expected data in some kind of internal representation, not text. Being able to generate data in *any* kind of internal representation means that we are able to benchmark specific components, saving computational resources.

A natural approach to object graph generation is to reuse our existing grammar-based text generator, write the grammar of a graph description language, generate descriptions of graphs, and then build the corresponding graphs.

To do so, we have targeted STON, a superset of JSON specifically designed to describe object graphs. It permits object graphs with cycles (contrary to JSON which only produces forests of anonymous objects), as well as creating instances of specific classes. By creating grammars that generate subsets of STON, one can easily create graphs that are valid inputs of the benchmarked program.

3. Guided benchmark generation

While our input generation is very similar to how it works in other fuzzers, the detection of performance bugs is significantly different from the detection of a behavioral bug. Indeed, due to the lack of specifications for a software's performance, we cannot decide if a given execution of a program on a certain input reveals a performance issue or not.

To solve this issue, modern fuzzers come with an oracle that is able to distinguish between sound executions and bug-exhibiting executions, and use it to guide generation [8] [9]. These oracles, however are not designed to detect performance issues. We therefore have to investigate the following questions

- how to detect a performance issue ?
- how to use feedback information as an oracle to guide generation?

3.1. Evaluating a score for each sample

Due to the lack of performance specifications, we avoid trying to classify executions between those that exhibit performance issues, and those that don't. Instead, we measure how poorly has a certain execution gone, from a performance perspective. This allows us to turn our problem into an optimization problem: we try to find the inputs that make the program perform the worst. We then report it to the programmer, who judges whether the found worst case should be optimized or not.

A natural metric for measuring performance is measuring the execution time of the program. However, accurately measuring the execution time is hard because there is an important amount of noise. This noise is due to several factors that are hard to control, among which the hardware and the operating system, the other processes running in parallel, the garbage collector, the JIT compiler and Pharo's scheduler. This makes time measurement very unstable (measuring the same execution has a lot of variance) and harder to reproduce. This hinders feedback information because it leads to wrong conclusions. The only way around is to repeat the measurements, to get statistically significant results [10], but this is expensive, and doesn't address systemic measurement biases (such as the machine on which the experiments are run).

To avoid the aforementioned pitfalls, we have exploited the fact that, in pure OO languages, most of the computation is done by sending messages. Indeed, the number of message sends, in Pharo, is much more stable than the duration of the execution, yet very strongly correlated with the average execution time and broadly machine-independent [11].

In order to count the number of message sends without installing a full profiler, which needs to wrap every method and is therefore very expensive, we modified the virtual machine[3] to increase a global counter each time a method is called. Additionally, we added a primitive to retrieve and reset that counter and, based on that, we provide a custom profiler `GncProfiler`. On all the experiments that we have run, this instrumentation incurred an overhead of at most 300%.

[3] `https://github.com/jthulhu/pharo-vm/tree/feature/blop`

From now on, whenever we mention counting the number of message sends, we implicitly mean that they have been counted using our profiler. If the user provides two programs to be comparatively benchmarked (for instance, two versions of the same software, to find performance regressions), the computed score is the ratio between the number of message sends. Otherwise, the score is simply the number of message sends.

3.2. Using feedback information for guidance

Because we only have access to a score for each sample, and not more fine-grained metrics, we perform black-box feedback guiding using general-purpose optimization algorithms. The metaheuristics we have used are simulated annealing and genetic optimization, where the individuals are the grammars alongside the weights of their rules. Each grammar is identical, but the weights are the parameters that the metaheuristic tweaks. A score is associated to each individual by sampling it, that is, generating an input data, then computing its score as described above.

3.2.1. Simulated annealing

One of the metaheuristic that we have implemented for our benchmarking framework is simulated annealing [12] [13]. The idea is to start with a randomly generated grammar, then mutate it slightly (here, we mutate one of its parameters at random). If the newly created individual is better than the original one, then it becomes the current individual, as in a regular hill climbing algorithm. Otherwise, we choose between the original individual and the mutated version. The probability for the younger individual to be chosen depends on how much worse it is with respect to the previous individual, and on the temperature, which is an additional parameter of the algorithm. The higher the temperature, the higher the probability to switch to the younger individual, even if it is worse. This means that, at higher temperatures, it is easier to escape local maxima. On the other hand, at higher temperatures, the algorithm won't converge towards a good solution. For this reason, the temperature is progressively lowered.

There are many possible variations on this schema, including choosing how and when to decrease the temperature, an initial choice of temperature, and the actual probability formula. In our implementation, the temperature follows an inverse law $T = \frac{T_0}{t}$, where T_0 is the initial temperature and t represents the advancement of the algorithm. The probability for transitioning to a worse individual is an exponential law $e^{\frac{\Delta}{T}}$, where Δ is the score difference. The initial temperature is left as a parameter dependent on the actual program we want to benchmark.

3.2.2. Genetic optimization

The other metaheuristic available in our benchmarking framework is genetic optimization, which is also the metaheuristic used by AFL [14], a fuzzer that has successfully found a very high number of critical bugs [15].

The core idea of genetic optimization is to keep a diverse population of individuals, that is incrementally improved by combining the individuals to produce new individuals.

The size of the population being fixed, the newly produced individuals replace the old individuals, creating a new generation.

To combine two (or more) individuals to create an offspring, genetic optimization algorithms rely on two mechanisms: mutation (a parameter is randomly assigned to a new value), and crossover (a parameter or a sequence of parameters in the offspring is copied from either parents, at random).

Based on this scheme, several different implementations exist. We have hand tuned our implementation by testing it on a toy problem that is meant to be easily solved by genetic optimization algorithms. In this problem, individuals are fixed-length string. Their score is the number of indices where they match a predetermined string. The goal is to find that string, by trying to find a string that maximizes its score. This lead us to the following implementation:

- the population has a fixed size of 100;
- at each generation, the 10% fittest (with highest score) individuals are passed to the next generation as-is;
- the rest of the new population is filled by sequentially choosing two parents among the 50% fittest individuals, and combining them;
- when two parents are combined, each parameter of the offspring is either randomly chosen from a parent, or is assigned a random value.

4. Experimentation

In this section, we are going to compare several approaches to fuzz benchmarking. We are going to assess whether fuzzing can be used to find performance bugs, and how much guiding can improve naive random fuzzing. To do so, we are going to compare two regular expression engines, matching simple regular expressions on the smallest string they actually match. The first regex engine is the one from Pharo's standard library, and uses a backtracking algorithm, which is relatively fast in general, but has an exponential complexity in the worst case. The second regex engine is one we developed for the experience. It uses Thompson's construction to build a non-deterministic finite automata with ε-transitions [16]. It then does on-the-fly simulation of the powerset algorithm to match the regular expression. This algorithm has a linear complexity.

For each regular expression r, we will compute a score s_r for that regular expression. That score is defined as $\frac{s_{r,b}}{s_{r,a}}$, where $s_{r,b}$ is the number of sent messages matching r on the smallest string it matches, using the backtracking algorithm, and $s_{r,a}$ is the same but using the automata algorithm.

Furthermore, since we want to identify regular expressions that exhibit a bug, we are going to consider a threshold regular expression: it's a handcrafted regular expression, which has been specifically thought has an example of poor performance for backtracking algorithm: $a?^n a^n$ [17]. For our purposes, we are going to consider regular expressions of

size at most 40, so the threshold regular expression is $r_t := a?^{14}a^{14}$. We will consider that a regular expression r reveals a performance bug if its score s_r is greater or equal to the threshold score $s_{r_t} = 159$. Indeed, that would mean that we found a regular expression that makes the regex engine perform *worse* than the purposefully handcrafted one.

Furthermore, we assume that we have a computational budget of 10000 samples, which means that we imagine that a programmer trying to find bugs would let our infrastructure try 10000 samples (which takes between 10 and 20 seconds on our machine): if a bug is not found within that computational budget, then (for this experiment) we consider that finding a bug is too expensive, and therefore that we have failed at finding one. Hence, for each technique, we will run 30 independent batches of 10000 samples each. Then, we will analyze three aspects of these batches:

- **Consistency.** How often is a bug revealed? This is ratio of batches which have produced at least an example which exhibits a bug over total number of batches.
- **Specificity.** How well do the examples that exhibit a bug specifically exhibit the behavior that leads to poor performance? We assume that a higher score means an input that highlights better what kind of input triggers a performance issue. For someone trying to find such a bug, an input where only half of it is relevant is less interesting than an input whose components are all relevant in triggering the bug.
- **Inevitability.** For the batches that have failed, did they fail because the meta-heuristic did not converge fast enough towards a bug (and therefore if we increased the computational budget, we would have found the bug eventually), or did the technique completely miss the bug? This is a qualitative criteria.

We will *not* analyze the diversity of the batches, because in our simple example we expect that the implementation algorithm will outweigh any other performance issue; on real programs, however, we expect that several performance bugs coexist, and therefore a technique that finds diverse bug-exhibiting examples might actually find more than a single bug per batch.

4.1. Unguided generation

A first approach is to define a grammar of the wanted input, generate lots of inputs, and check there are any outliers. The first research on fuzzing started with completely random input generation [1].

4.1.1. Experimentation results

Figure 3 shows the score of the best individual of each batch. The maximum score reached is about 360, and the second best is 140.

Consistency. This means that exactly one of these 30 runs successfully identified a performance bug, according to our criteria. While this indicates that fuzzing can be used to find performance bugs, even without any guidance, the consistency of this technique is very low.

Specificity. The only example that exhibits a performance issue, however, has a score that is more than twice the threshold, so it is more specific than the handcrafted example.

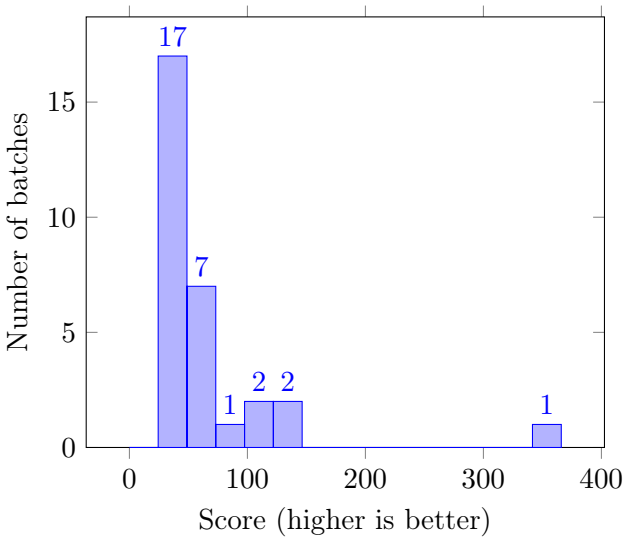

Figure 3: Score distribution of the best examples in each batch

Inevitability. Finally, since running 30 batches of 10000 samples each is the same as running a single batch of 300000 samples, each batch will likely eventually discover a performance bug. If the input space were much bigger, however, the time it would take to eventually discover a bug might become prohibitive.

An issue with this approach is revealed by taking a look at how the score distribution of the sampled regular expressions, which can be seen in Figure 4. Approximately 90% of them are within a factor two of the other. A smaller fraction (9%) of regular expressions are matched between 5 and 15 times faster by our implementation. The remaining 0.34% of the cases are outliers that perform significantly better with our implementation.

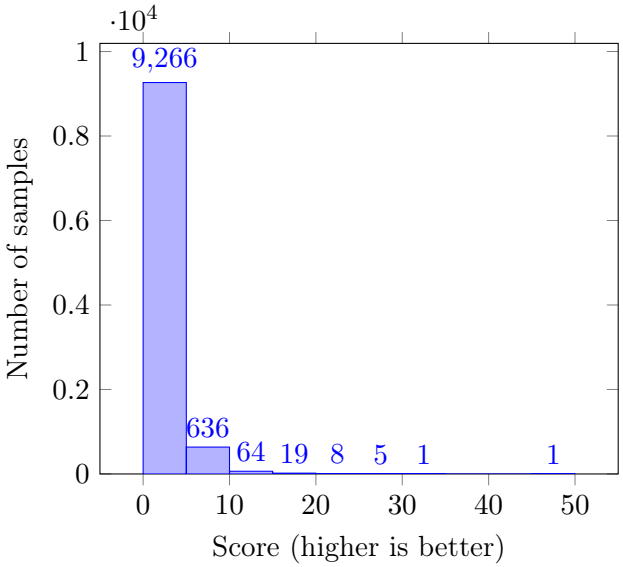

Figure 4: Typical score distribution of an unguided generated batch

4.2. Simulated annealing

The analysis above shows that, despite being able to find some interesting outliers, we spend of our time generating samples that are uninteresting, without reusing information we gained about them. Instead, at each sample generation, we could use information gained in previous iterations to guide the generation. This is typically what simulated annealing does.

4.2.1. Simulated annealing results

In Figure 5 are shown the results of the experiment, with different initial temperatures Each boxplot corresponds to the aggregation of the best score of 30 runs, each testing at most 10000 regular expressions. The first boxplot, in blue, show, for comparison, the data collected by unguided random sampling, presented in the same way to ease comparison.

Specificity. With initial temperatures under 40000, our runner finds an example which runs significantly slower in Pharo's implementation. Above that temperature, in the worse case it fails to find a performance issue, on the contrary to random sampling which consistently finds examples which run at least 12 times slower on Pharo's implementation.

However, we also note that the simulated annealing has more widespread results. Even at higher temperatures, some outliers give better results than random sampling. At the initial temperatures that overall seem to perform better (between 5000 and 20000), some outliers are over five order of magnitudes slower in Pharo's implementation, with the peak being at 228043. This means that simulated annealing has a much higher specificity than random sampling. An example of a very specific regular expression generated with this method, that has a score over 160000 is `b*((b|(b|b|b)))+b+(b)*b+b+b+b+b+(bb+b+)+`.

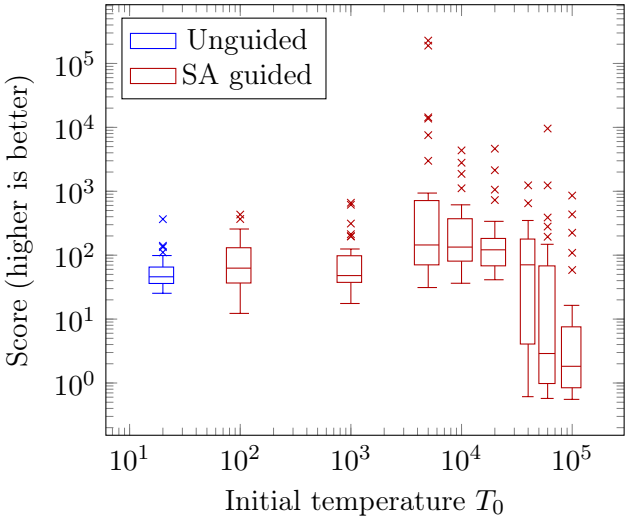

Figure 5: Simulated annealing results, by initial temperature T_0

Consistency. Furthermore, the batch with the best initial temperature ($T_0 = 5000$) successfully finds a performance bug 14 times out of 30. It is significantly more consistent than random sampling.

Inevitability. For higher initial temperatures, however, the big problem is inevitability: for $T_0 = 100000$, 9 out of 30 executions fail to converge to an example that has a score greater than 1, that is, an example that runs faster on our implementation! The initial temperature is too high, and therefore we would have to wait too long to eventually find a useful example.

4.2.2. Discussion

We think that the instability of the simulated annealing is due to a difficulty of getting out of local maxima. Indeed, when the execution is "lucky enough" not to get in a local maxima, or it manages to get out, with a reasonable initial temperature, it reaches very good solutions, but on a significant number of batches it performs worse than random sampling because it gets stuck. Since simulated annealing's purpose is to avoid getting stuck in local maxima, we believe that a finer tuning of our work would greatly improve consistency.

4.3. Genetic optimization guiding

One of the defects of the previous approach is that it lacks diversity. If the initial solution is unlucky and gets stuck in a pretty bad local maximum, then the whole run will most likely not result in an interesting finding. A metaheuristic that is less affected by this problem is genetic optimization, because it starts with hundreds of different individuals, and at each step the parameters can be scrambled much more. For each batch, after generating 100 random individuals, we run for 100 generations.

4.3.1. Genetic optimization results

In Figure 6, the GO guided samples are compared with the unguided ones, and with the best SA guided samples.

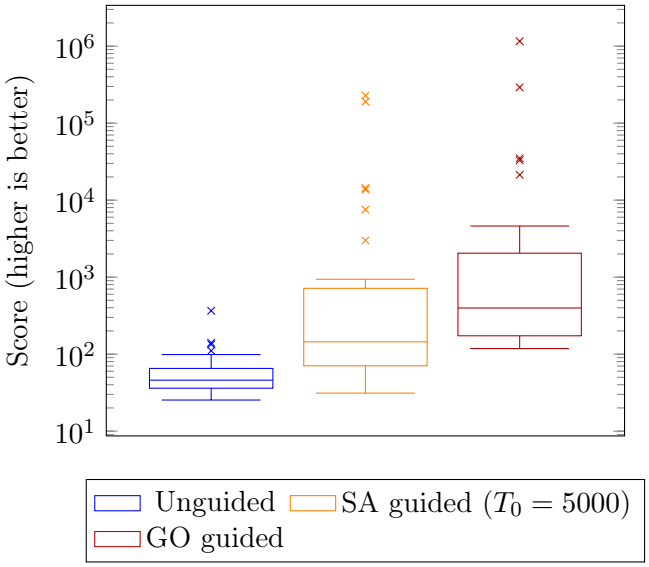

Figure 6: Genetic optimization results

Consistency. First of all, we see that, indeed, this method is more consistent than random results: 25 runs out of 30 have successfully detected a bug, according to our initial criteria, so it is even more consistent than simulated annealing.

Specificity. However, this is not the only improvement. As can be seen on Figure 6, genetic optimization guidance seems to produce more specific samples, with a peak example, `(a|a|a|a|a|a|c)+a?a+.(a)+(a+a+a+(a)+a)b`, whose score is over $1.1 \cdot 10^6$!

Inevitability. Finally, genetic optimization's worst results are above the best results of random generation (without taking into account the outliers), meaning that it is very

likely that with a little more computational budget, even those batches would have caught up.

We believe that genetic optimization outperforming simulated annealing in our experiments is partly due to the finer tuning our GO framework has been through, and partly due to the fact that our currently implementation of genetic optimization keeps more diversity among its samples.

5. Conclusion

We explore whether fuzzing techniques can be applied for automatic benchmark generation, based on the insight that automatic benchmark generation and test generation are similar problems, and that fuzzing is a known proficient solution for the latter. We show that input generations techniques can be transposed directly to benchmark generation, but the feedback collection step, that allows guiding of the input generation, is significantly different.

Figure 7: Comparison of the provided metaheuristics

Guidance	Consistency	Specificity[4]
Unguided	1/30	2.3
SA ($T_0 = 100$)	5/30	2.7
SA ($T_0 = 1000$)	6/30	4.2
SA ($T_0 = 5000$)	14/30	1434
SA ($T_0 = 10000$)	12/30	27.4
SA ($T_0 = 20000$)	10/30	29.2
SA ($T_0 = 40000$)	10/30	7.8
SA ($T_0 = 60000$)	5/30	60.3
SA ($T_0 = 100000$)	3/30	5.4
Genetic Optimization	25/30	7280

We provide a framework for automatic benchmark generation based on a general-purpose grammar-based data generator. This generator can be directed towards certain input, and is agnostic of the metaheuristic used for this guiding. Furthermore, we also provide two hand tuned metaheuristic that can simply be plugged in to guide the input generation based on certain metrics, but they also are agnostic on the underlying metric. Finally, we provide a VM-level instrumentation to measure efficiently a reliable proxy for execution time.

A summary of the results of the experiment we have conducted are shown in Figure 7. In our setup, simulated annealing performs better than no guidance at all for some initial temperatures, and genetic optimization outperforms simulated annealing and no guidance.

In the future, other metrics (such as code coverage, execution trace coverage, memory usage, number of allocations) could also be measured in place of execution time, and other metaheuristics could be implemented. Besides, rather than using black-box fuzzing techniques, one could use symbolic or concolic execution to target specific code spans, which is particularly useful in case we want to ensure a given patch doesn't introduce regressions.

[4]Ratio between the best score s_r and the threshold score s_{r_t}.

Appendix

A. Regular expression grammar

For the sake of simplicity, we define a grammar generating a sufficiently powerful subset of regular expressions, which includes characters, the wildcard pattern match `.`, concatenation, alternative `|` and grouping, in addition to usual modifiers such as `+`, `*` and `?`.

Originally, the grammar was the following.

```
defineGrammar

  ntRegex --> ntConcat | ntConcat, '|', ntRegex.
  ntConcat --> ntModifier | ntModifier, ntConcat.
  ntModifier --> ntAtom
               | ntAtom, '?'
               | ntAtom, '*'
               | ntAtom, '+'.
  ntAtom --> 'a' | 'b' | 'c' | '.' | '(', ntRegex, ')'.

  ^ ntRegex
```

However, this didn't work well because Pharo's implementation does not support applying modifiers to regular expressions that can match the empty string. Luckily enough, context-free grammars are powerful enough to describe regular expressions that do not match the empty string, at the cost of being a little bit more verbose.

```
defineGrammar

  ntRegex --> ntConcat | ntConcat, '|', ntRegex.
  ntConcat --> ntModifier | ntModifier, ntConcat.
  ntModifier --> ntAtom
               | ntNonEmptyAtom, '?'
               | ntNonEmptyAtom, '*'
               | ntNonEmptyAtom, '+'.
  ntAtom --> 'a' | 'b' | 'c' | '.' | '(', ntRegex, ')'.
  ntNonEmptyAtom --> 'a' | 'b' | 'c' | '(', ntNonEmptyRegex, ')'.
  ntNonEmptyRegex --> ntNonEmptyConcat
                    | ntNonEmptyConcat, '|', ntNonEmptyRegex.
  ntNonEmptyConcat --> ntNonEmptyAtom
                     | ntNonEmptyAtom, ntConcat
                     | ntModifier, ntNonEmptyConcat.
  ^ ntRegex
```

References

[1] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," vol. 33, no. 12, pp. 32–44, Dec. 1, 1990.

[2] B. Miller, D. Koski, C. Lee, *et al.*, "Fuzz Revisited: A Re-Examination of the Reliability of UNIX Utilities and Services," Jan. 1, 1998.

[3] S. Van Caekenberghe. "Smalltalk Object Notation (STON)." (Feb. 14, 2012), [Online]. Available: `https://github.com/svenvc/ston/blob/master/ston-paper.md` (visited on 07/21/2023).

[4] S. Zaman, B. Adams, and A. E. Hassan, "A qualitative study on performance bugs," Jun. 2012, pp. 199–208.

[5] K. V. Hanford, "Automatic generation of test cases," vol. 9, no. 4, pp. 242–257, 1970.

[6] P. Purdom, "A sentence generator for testing parsers," vol. 12, no. 3, pp. 366–375, Sep. 1, 1972.

[7] W. H. Burkhardt, "Generating test programs from syntax," vol. 2, no. 1, pp. 53–73, Mar. 1, 1967.

[8] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed Greybox Fuzzing," Dallas Texas USA: ACM, Oct. 30, 2017, pp. 2329–2344.

[9] M. Christakis, P. Müller, and V. Wüstholz, "Guiding dynamic symbolic execution toward unverified program executions," Austin Texas: ACM, May 14, 2016, pp. 144–155.

[10] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically Rigorous Java Performance Evaluation,"

[11] A. Bergel, "Counting Messages as a Proxy for Average Execution Time in Pharo," in M. Mezini, Ed., vol. 6813, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 533–557.

[12] M. Pincus, "Letter to the Editor—A Monte Carlo Method for the Approximate Solution of Certain Types of Constrained Optimization Problems," vol. 18, no. 6, pp. 1225–1228, Dec. 1970.

[13] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," vol. 220, no. 4598, pp. 671–680, May 13, 1983.

[14] M. Zalewski. "American Fuzzy Lop." (Nov. 12, 2013), [Online]. Available: `https://lcamtuf.coredump.cx/afl/` (visited on 07/26/2023).

[15] M. Zalewski. "Bugs found by AFL." (), [Online]. Available: `https://lcamtuf.coredump.cx/afl/#bugs` (visited on 07/28/2023).

[16] K. Thompson, "Programming Techniques: Regular expression search algorithm," vol. 11, no. 6, pp. 419–422, Jun. 1, 1968.

[17] R. Cox. “Regular Expression Matching Can Be Simple And Fast.” (Jan. 2007), [Online]. Available: https://swtch.com/~rsc/regexp/regexp1.html (visited on 07/26/2023).