Dynamic Fitness Functions for Genetic Improvement in Compilers and Interpreters

Oliver Krauss  
Institute for System Software - ISS  
Johannes Kepler University Linz  
Linz, Austria  
Advanced Information Systems and Technology - AIST  
University of Applied Sciences Upper Austria  
Hagenberg, Austria  
oliver.krauss@fh-hagenberg.at

Hanspeter Mössenböck  
Institute for System Software - ISS  
Johannes Kepler University Linz  
Linz, Austria  
hanspeter.moessenboeck@jku.at

Michael Affenzeller  
Heuristic and Evolutionary Algorithms Laboratory - HEAL  
University of Applied Sciences Upper Austria  
Hagenberg, Austria  
michael.affenzeller@fh-hagenberg.at

ABSTRACT
When attempting to improve the non-functional requirements of software, specifically run-time performance of code, an important requirement is to preserve the correctness of the optimized code. Additionally when attempting to integrate Genetic Improvement into a compiler or interpreter, the large search spaces resulting from the amount of operators and operands a language provides needs to be dealt with. This publication explores dynamic fitness functions as a foundation for a use in Genetic Improvement to optimize programs. An approach of using a test suite to verify code correctness in the Truffle Framework [19, 20] and Graal Compiler [11] is presented. Two types of fitness functions are explored, which split the test suite according to their complexity and attempt to generate correct solutions with a growing set of increasingly complex tests. One of them increases the amount of tests sequentially over several iterations. The parallel fitness function attempts to split a test suite and to re-combine the results with increasingly large suites. The results show that these functions only marginally improve the fitness landscape on their own, but that more partially correct solutions can be found with dynamic fitness functions. In the future, our approach may be improved by implementing specific crossover and mutator operations to accompany the dynamic fitness functions.

CCS CONCEPTS
- Software and its engineering → Source code generation; Interpreters;  
- Computing methodologies → Genetic algorithms;

KEYWORDS
Genetic Improvement, Fitness Functions, Test Driven Verification, Test Complexity

ACM Reference Format:

1 INTRODUCTION
When creating a new programming language or implementing a new execution environment (such as an interpreter or a compiler) for an existing language, one challenge is to achieve sufficient performance to be competitive with other existing environments. When attempting to improve the non-functional requirements of code, such as run-time performance or memory efficiency, a crucial requirement is to preserve code semantics.

When Genetic Improvement (GI) is used to optimize programs, a fitness function must be selected that can satisfy the constraint of code correctness while improving on additional non-functional aspects. The problem here, is that the fitness landscape of such a fitness function has a significantly lower average quality, as well as bigger gaps between qualitative-peaks, compared to fitness functions that optimize only towards code correctness or performance.

This publication, which is part of an ongoing research series [7], describes an attempt to introduce a dynamic fitness function for GI in a modern interpreter and compiler framework, which is provided by Truffle [19] and Graal [11] as part of the OpenJDK project. The core research question of the series is to find optimizations using GI that improve run-time performance and provide benefits beyond already existing compiler optimizations. The research concentrates on additional optimizations; replacing existing ones is not our goal. The challenge this publication addresses is to improve fitness functions, and with them the fitness landscape, over complex search spaces. These search spaces are the result of the combination of run-time performance with code correctness, as well as the large search space in a given language (amount of operators and operands).

The reason for using dynamic fitness functions is to move away from a static fitness landscape with larger gaps, to a dynamic landscape with a fitness function that continuously increases the pressure on the requirement of code correctness. This is done by splitting the correctness metric into several smaller more manageable parts. In the context of this work, code correctness is measured...
by using test cases, as is commonly done in the GI and genetic algorithm literature [3, 8, 14]. These test cases are evaluated for their complexity, and then split into groups which are applied to the fitness with increasing complexity.

The remainder of this publication is split into the following parts:

- **Section 2** details the architecture on which this work is based and gives a short introduction on the Graal compiler and the Truffle interpreter.
- **Section 3** measures the test complexity and details on the dynamic fitness function can be found in Section 3.
- **Section 4** shows four simple programs are shown in Section 4.
- **Section 5** interprets these results and describes possible threats to validity.
- **Section 6** discusses further research directions are discussed in Section 6.

## 2 TECHNICAL BACKGROUND

The following section gives an overview of the Graal Compiler, the Truffle Interpreter, and the MiniC Language which is written in Truffle and executed on the Graal Virtual Machine (VM). Additionally, a short overview of how GI is used to optimize the MiniC language is given. In the context of this work Graal, Truffle and MiniC are used as follows:

- **Graal** is used as compiler for MiniC. It is a highly optimizing JIT compiler that is available in OpenJDK since Java 9 and is used to compile Truffle ASTs from bytecode to machine code.
- **Truffle** is the interpreter framework that MiniC was written in. It can interpret MiniC on the JVM without the use of Graal, albeit with a lower performance.
- **MiniC** is an experimental language written in Truffle and compiled on Graal.

The technology stack was selected as Truffle and Graal allow rapid prototyping of a programming language. Additionally, the AST structure of Truffle enables easy manipulation of a given AST [19, 20]. These reasons make them suitable for GI-related research.

### 2.1 Graal

Graal [11] is an aggressively optimizing just-in-time (JIT) compiler, written in Java as part of the OpenJDK project. It compiles Truffle ASTs (in bytecode) to efficient machine code and features several optimizations, including speculative optimizations which can be taken back if necessary (called deoptimization). Graal uses an IR that is a directed graph describing both control flow and data flow. The Graal IR can be viewed with the Ideal Graph Visualizer (IGV).

Both Graal and IGV are open source and build a basis for research into compiler optimizations. [16, 17]

### 2.2 Truffle

Truffle [19] is a self-optimizing interpreter framework for implementing new languages based on Abstract Syntax Trees (AST). Truffle itself does not feature a lexer, parser or linker. The focus relies solely on implementing AST nodes that are combined into a Truffle language that can be executed on any Java VM. Every

## 2.3 MiniC

MiniC is a subset of the C11 version [5] of C. The features that are most notably missing from the language are, at the time of writing, structs, pointers and unsigned data types. Even though the language is so limited it consists of 342 node classes of which 55 are terminals. All of these nodes may be used as operators or operands in a GI run. The language is experimental, and was primarily written to test and improve integration of GI into the Truffle framework. A Coco/R-generated parser [10] is used to transform MiniC code into Truffle AST nodes. All experiments presented in this publication were run in MiniC.

### 2.4 Genetic Improvement in Graal and Truffle

An algorithm suite and an experiment framework were implemented as an additional module to the Truffle framework, as can be seen in Figure 1. At the core of the framework stands the Truffle Language Analyzer (TLA). Using the current ClassLoader the TLA collects and analyzes all existing AST nodes in all packages or classes given by an implementer into the Truffle Language Information (TLI). AST nodes are identified by checking if they implement the Node class provided by Truffle. The TLA also provides some analytical features for a given language. One of them is the full-analysis of (smaller) landscapes to gain insights about the language and the nodes selected for experimentation.

### Listing 1: Example Truffle Implementation

```java
@NodeChildren(
    @NodeChild("leftNode"),
    @NodeChild("rightNode")
)

public abstract class MinicIntAddNode extends MinicIntArithmeticNode {
    public int add(int left, int right) {
        return left + right;
    }
}
```
The purpose of the TLI is to provide information on nodes of a language in a generalized way. It contains information on which nodes are terminal nodes, which nodes are non-terminal and how many child nodes are necessary at minimum to reach a terminal from them. Additionally, a mapping from each abstract node (represented as an abstract class) to all found implementations of that node is provided. An example of an abstract node can be seen in Listing 1 with the classes MinicIntArithmeticNode and MinicIntAddNode. Truffle auto-generated the specific class MinicIntAddNodeGen below the generic classes. Finally, the TLI contains an initialization mechanism for each non-abstract class as each Truffle Node can be instantiated by only one of the following three methods:

- A given public constructor exists only on non-abstract classes created by an implementer
- A create method that is auto-generated from Truffle nodes for abstract classes (as seen in Listing 1)
- A node factory that is auto-generated by Truffle with the GenerateNodeFactory annotation

The TLI is used by the Optimizer, which contains features to analyze an AST based on given tests, providing an Input/Output comparison as well as complexity metrics. Additionally, the runtime is analyzed and a RuntimeProfile is created containing statistical information such as the Median, Quartiles, minimal and maximal runtimes, and the amount of executions in each quartile. A connector to HeuristicLab [6], a framework for heuristic and evolutionary computation with a strong focus on genetic programming and containing a multitude of genetic operators, is currently under development.

One advantage of using GI with Truffle and Graal is the use of AST nodes that are directly interpreted and/or compiled, as well as the information about the Stack and Heap provided in the form of Frames and Materialized Frames by the Truffle framework. This information can be leveraged when creating and modifying ASTs, which eliminates the problem of accessing variables out of scope. This removes one of the primary reasons for uncompilable ASTs observed in literature [9].

3 METHODOLOGY

The following section explains how a test-driven approach to GI can be split into more manageable parts, evolving increasingly complex ASTs to increasingly satisfy the constraint of code correctness.
The approach of increasing test complexity iteratively has been previously explored in Code Synthesis with a DSL [12, 13].

3.1 Test-Driven Fitness Evaluation

A test driven approach to software verification cannot guarantee semantic equivalence of two different ASTs. However, a sufficiently high confidence in the correctness of generated ASTs can be achieved with a sufficiently qualified test suite [15]. For the purpose of this publication, a sufficient test suite is one that has full coverage of all branches on an AST selected for optimization.

In the presented framework, the test data is provided manually by an implementer in the form of IN and OUT files. Each of these files contains one test per line, each test consisting of all required input parameters or expected output, separated by a comma. For each value the required data type is given as well.

Figure 2 shows the process of the optimization. After parsing a program into MiniC nodes and selecting an AST for optimization, the tests, provided by an implementer, are executed and verified directly with the Truffle Framework. This step serves to verify the correctness of the given tests themselves. The tests are then reused in the evaluation phase of the genetic programming (GP) algorithm for all compilable ASTs. A test counts as succeeded if given an input, its output matches the given output. If the AST produces no output or an output of a different data type it does not count as successful. Thus a normalization of the output for comparison is not necessary.

The correctness is calculated as follows:

$$\text{correctness}_{(\text{AST, tests})} = \frac{\sum_{n=1}^{\text{tests}} \text{succeeded}_{(\text{AST, test})}}{\sum_{n=1}^{\text{tests}}}$$

3.2 Evaluating Test Complexity and Overlap

To sort tests by their complexity, they need to be assigned a complexity value. The current approach is to measure the amount of nodes that are visited with a given test case. The type of the node itself (Operator, Operand, FunctionCall, ...) is not considered to be part of the test complexity:

$$\text{complexity}_{(\text{AST, test})} = \frac{\sum_{n=1}^{\text{nodes}} \text{visited}_{(\text{AST, test})}}{\sum_{n=1}^{\text{nodes}} (\text{AST})}$$

Figure 3 shows an example of the complexity metric with an AST that represents a recursive algorithm for calculating a value in the Fibonacci sequence. The given input $n$ produces the $n$th number of the sequence. When given the input 0, 14 of the nodes are activated, meaning that the test case is assigned a complexity of 14/24. An input of 1 has a complexity of 11/24 as it activates all nodes as assigned in the figure except the 3 ($n == 0$) nodes, since the $||$ expression already becomes true with ($n == 1$). An input of 2 or higher would activate all nodes except the (ret n) nodes, as the activation in the recursive call is not counted, in the if path of the algorithm, meaning that all tests with an input of 2 or higher would be assigned a complexity of 22/24.

This approach makes the complexity value of a test case dependent on the given implementation of an AST that is considered for optimization. For example, if the AST in Figure 3 was implemented with $(n < 2)$ as a condition for the if statement then the complexity of both test cases with input 0 and 1 would be equal with a complexity of 10. However, this metric does allow the automation of determining test case complexity, and still allows a comparative view of ASTs with control structures. In an AST without control structures, such as mathematical expressions, every test case has the same complexity, meaning that dynamic fitness functions have some problem areas where they will not provide an advantage, as the complexity metric is equal for every test.

Another metric of interest is the overlap between test cases. The overlap is defined as the amount of nodes that two tests both visited compared to the maximum of visited nodes:

$$\text{overlap}_{(A, B)} = \frac{\sum_{n=1}^{\text{nodes}} \text{visited}_{A} \cap \text{visited}_{B}}{\max(\sum_{n=1}^{\text{nodes}} \text{visited}_{A}, \sum_{n=1}^{\text{nodes}} \text{visited}_{B})}$$

If two test cases have a complete overlap (1), this can be an indicator that one of the test cases can be removed from the test set as it
tests the same functionality with different data. When considering the alternative implementation to Figure 3 with \( n < 2 \) as condition, the tests with input 0 and 1 would have an overlap of 1. In the given implementation, however, their overlap is 0.786. Again, the overlap value depends on the provided AST, and while a complete overlap indicates candidates for removal, not all overlapping tests should be removed, as this would make the test suite entirely dependent on the specific AST it was evaluated on.

Another metric that can be gained by analyzing the impact of test cases on AST execution is the confidence score. The confidence score represents the sum of all visited nodes over all test cases:

\[
\text{confidence}(\text{testsuite}, \text{AST}) = \frac{\sum_{n=1}^{\text{nodes}} \text{visited}_{\text{testsuite}}(n)}{\sum \text{nodes}(\text{AST})}
\]  

If all nodes are visited in a test suite this does not still guarantee absolute semantic equivalence between two ASTs when they both successfully execute the suite. It is, however, an indicator that can be used to increase the confidence that at least all paths of an original AST were considered when optimizing it. The confidence score also does not correlate to code stability. When considering Figure 3 any test with a negative input would result in a stack overflow due to infinite recursion. This is due to the fact that the given definition is equivalent to code coverage, but does not consider test adequacy.

### 3.3 Dynamic Fitness Functions

Two distinct versions of a dynamic fitness function were created. The sequential version increases the amount of test cases used in the fitness function of a GA continuously. The parallel version splits the problem into different test groups with tests selected by overlap (Equation 3).

#### Sequential Fitness Function

The sequential fitness function / algorithm as shown in Figure 4, sorts a given test suite by increasing complexity (Equation 2) and splits it into several approximately equal parts (5 Tests -> 3 Groups results in 2|1|2). The amount of parts is defined by the implementer, but is at most the amount of tests given.

After splitting the suite, the first part, containing the tests with the lowest complexity, is evaluated by a regular genetic algorithm. After evaluation, the sequential algorithm combines the second test group with the first group and the best \( n \) members of the final population in the GA are selected and injected as elites of the first generation of the next run. This process is repeated until the last GA runs with the entire test suite, as depicted in Figure 4.

#### Parallel Fitness Function

The parallel fitness function / algorithm, shown in Figure 5, splits a test group into several equal parts. The split is conducted by randomly assigning one test to each empty group. From then on, using round robin, each group is assigned the test with the highest overlap (Equation 3) of the remaining unassigned tests until all tests are assigned.

Each of the groups is then evaluated using a genetic algorithm. After evaluation, the parallel algorithm reduces the amount of groups, and the tests from removed groups are distributed through round robin to the remaining groups. The remaining groups are then evaluated using the genetic algorithm again, each with \( n \) members of the final populations of all groups that provided tests injected as elites into the new group. This process is repeated until the last GA runs with the entire test suite as one group.
3.4 Use Cases and Configuration

The use cases used for the results are as follows:

- **x^2** is a simple test returning twice the input value. It is used as baseline to compare with the other tests.
- **Sqrt** calculates the square root of a given floating-point number down to 0.000000001 exactness.
- **Fibonacci** returns the positive range of the Fibonacci sequence.
- **Constructed** a constructed case, as seen in Listing 2 that has no real world use, but was created to enable tests with varying complexity (Equation 2).

### Table 1: Use Case statistics, including the number of nodes produced by the parser, the amount of tests used in the evaluation, and the number of nodes visited by each test

<table>
<thead>
<tr>
<th>Use Case</th>
<th>No. Nodes</th>
<th>Tests</th>
<th>Nodes visited</th>
</tr>
</thead>
<tbody>
<tr>
<td>x^2</td>
<td>8</td>
<td>5</td>
<td>5, 5, 5, 5, 5</td>
</tr>
<tr>
<td>Sqrt</td>
<td>60</td>
<td>5</td>
<td>14, 42, 42, 42, 53</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>32</td>
<td>5</td>
<td>11, 15, 30, 30, 30</td>
</tr>
<tr>
<td>Constructed</td>
<td>31</td>
<td>5</td>
<td>11, 15, 20, 20, 21</td>
</tr>
</tbody>
</table>

### Listing 2: Constructed Use Case

```c
int constructed(int n) {
    if (n == 0) {
        return 1;
    }
    if (n == 1) {
        return 2;
    }
    if (n == 2) {
        return n + 1;
    }
    return n * n;
}
```

Table 1 contains some statistics about the tests for each of the given use cases. They list the number of nodes that the original AST for each test contained, the amount of test cases defined for each use case, and the amount of nodes visited per test, ordered by complexity (Equation 2).

To make the executions comparable all test suites have been evaluated with three different algorithms:

- **Parallel Fitness Function Algorithm** The parallel fitness function splits the problem description into three parts and recombines them into one part after the first execution.
- **Sequential Fitness Function Algorithm** The sequential fitness function splits the problem into four incremental parts, in order to have the same number of evaluations as the parallel fitness function.
- **Genetic Algorithm** A genetic algorithm without any modification was selected as the baseline. To make sure that it is comparable to the other algorithms, the genetic algorithm is restarted four times and carries over the same amount of best individuals into the next iteration. This was done to make sure that the continuous restarting with different populations, as done by the sequential and parallel algorithms, is also represented. Thus, the results should actually be comparable across the different types of fitness functions, and is not a result of restarting the algorithm, which re-introduces diversity into the population.

The configuration of the genetic algorithm used by all three evaluations is exactly the same. The values were selected after testing with different configurations:

- **Crossover** A single point crossover was selected.
When looking closer into the results, many solutions managed to the following results are based on several repeats of the test setup as x + 1, the algorithms get stuck in local optima and the solution for all values greater than 2 is never found. This does indicate that

4 RESULTS

The following results are based on several repeats of the test setup as described in Subsection 3.4. To show if the approach of dynamic fitness functions has any benefit, the primary indicator is the amount of runs that found a correct solution to a given problem.

The amount of found correct solutions is another indicator, but cannot be the primary one, as evolutionary algorithms naturally converge towards good solutions. This also means that if one correct solution is found, the likelihood of other solutions influenced by that solution in later generations increases. This is also proven by research into solution grafting, which improves the likelihood of finding valid solutions [1]. This was also a reason for excluding grafting operators from the experiment set.

Each use-case was tested on each algorithm group. Concerning the use cases:

As expected, the x*2 use case was solved in most runs of all algorithms. It was not expected that the parallel fitness function failed to solve any of the runs.

The Square Root function (sqrt) was not solved by any of the algorithms. When examining the results further, it was discovered that most runs managed to return approximations, with return 2 being the most-favored solution of all algorithms. The reason for this may be that of the tests had 4 as input, and this was the only test that was solved by this solution.

No algorithm managed to reproduce the Fibonacci sequence. When looking closer into the results, many solutions managed to return either one of the two first numbers in the sequence (primarily regular GA) or both of the first two numbers (primarily sequential fitness function).

For the constructed test case, also none of the algorithms manage to solve it. Since three test cases can be solved easily with returning x + 1, the algorithms get stuck in local optima and the solution for all values greater than 2 is never found. This does indicate that the incremental increase of the fitness function may also have a negative impact on the additional requirements.

Looking at the test results from the algorithms themselves, the regular GA solved slightly less runs than the sequential fitness function. This can be explained by a simple statistical outlier due to the low amount of runs. The sequential fitness function did, on average, manage to solve more test cases for each of the functions, including the x*2 function where the amount of nodes was equal for every test case. The reason for that is that in this case the first two tests selected for the first iteration had the inputs 1 and 2. Since one of these is solved by simply returning the literal 2, the following runs may mutate or cross-over this literal into the multiplication with the given input. Additionally the sequential fitness function finds on average more correct (Equation 1) solutions than the regular GA.

The parallel fitness function had the worst performance, not solving even a single execution in all test cases. Several of the first iterations in the parallel fitness function did manage to solve their subset of the test suite. However, no run was observed where all parts were solved at the same time. In addition to that, the final run of the parallel fitness function, which combined the best solutions of all previous sequences, often managed to produce an even worse result, failing previous succeeding test cases. This phenomenon was mostly observed in Fibonacci. It is to be noted that in these cases one of the succeeding solutions from the predecessor-generation, which did solve some test cases, remained the best solution until the end. This result is likely due to the used crossover operation which does not manage to construct sensible combinations of the previous results.

5 DISCUSSION

From the results it can be concluded that dynamic fitness functions, are not capable of largely extending the search space that can be used when attempting to use genetic improvement. Most applications of GI documented in literature [3, 8, 9] restrict the search space much further than we do, and the very low success rate can be attributed primarily to the amount of accepted operators/operands (103/55 respectively out of a total of 342 classes, of which most are abstract). When attempting to integrate GI with an interpreter or compiler, it is however important that the algorithm configuration needs as little restriction as possible.

The fact that both the sequential fitness function (overall) and the parallel fitness function (before the final combination of the test suite) solve more test cases on average than the regular GA shows that more time can be spent in improving the solution. Repeating the tests with a selection of sensible operators/operands as well as selecting sub-ASTs for modification may improve the results. Additionally, both functions could be improved with specific crossover/mutator implementations to accompany them. The biggest hindrance of the sequential fitness function is that it gets stuck in local optima, which could be solved by a mutator that specifically tries to extend the solution instead of replacing a subtree. Similarly, the problem of the parallel fitness function, which is also being stuck in one of the local optima from the previous iteration, this could be solved with a crossover that specifically attempts to merge the solutions using control structures (if statement, ...).
However, the results also show that using test cases in the fitness function itself is not useful in all instances, especially purely mathematical functions. The square root use-case could be solved with a fitness function using the value derivation for each test case instead of a simple solved / not solved per test case.

5.1 Threats to validity
One major threat to validity was identified during write-up of this paper. Unfortunately, the performance analysis performed for the publication measured all AST executions on the same JVM. This led to every AST being explored benefiting from the optimizations the Graal Compiler already performed on previous ASTs. Due to time restrictions the runs could not be repeated independently, as each AST analysis requires several thousand executions, for a useful performance analysis. The unusually high amount of executions (200,000) is specific to the presented approach, due to the use of the Graal Compiler. Graal has a warm-up phase for several of its optimizations which requires a correct performance measurement to ignore executions before the initial warm-up is completed.

As a result, the scope of this publication had to be reduced from performance optimization to code validity.

6 OUTLOOK
The methodology concerning the evaluation of test complexity (Equation 2) and overlap (Equation 3) already presented a confidence (Equation 4) scoring for a given set of tests. The shown methods and metrics can be considered to implement an algorithm that uses meta-heuristics for input fuzzing. In the case of an absent test suite, or to increase the confidence in a test suite, the automated generation of test cases would be an interesting approach for a further research direction. Building up on input fuzzing, collecting real-time execution data, similar to the approach in [3], could give insights into a scoring system for test cases that would order tests by the usage of the respective code paths. Thus, tests that cover rarely or never used paths could be assigned a lower score than tests that cover often used functionality. A fitness function based on code importance may enable further performance optimizations in non-critical environments.

The initial implementation of the sequential fitness function could be improved by using the Age Layered Population Structure (ALPS) GA [4]. The reason is that this enables a smoother transition between the increased amount of tests than restarting the algorithm in a new population.

The parallel fitness function could be implemented as an Island GP [18] algorithm where the amount of islands is gradually reduced. In addition to that, the parallel fitness function will require a specific crossover that uses control structures (such as if statements) to merge partial solutions.

ACKNOWLEDGMENTS
This publication is part of an ongoing doctoral thesis which is advised by Hanspeter Mössenböck and Michael Affenzeller.

REFERENCES