JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Oliver Krauss MSc

Submitted at
Institute for System Software

Supervisor and
First Evaluator
Prof. Dr. Dr. h.c. Hanspeter Mössenböck

Second Evaluator
Prof. Shin Yoo PhD

Linz, March 2022

Pattern Mining and Genetic Improvement in Compilers and Interpreters

Doctoral Thesis
to obtain the academic degree of
Doktor der technischen Wissenschaften
in the Doctoral Program
Technische Wissenschaften

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Acknowledgements

This work took me almost six years to create. In these years I learned much about the scientific process and the topics of this thesis, although I still feel like I just scratched the surface. During this time, many people supported me.

Hanspeter Mössenböck is an excellent supervisor, who always gave me swift and deliberate feedback. Michael Affenzeller likewise has provided me with needed guidance. Thank you both for your support throughout the years! I would also like to thank the members of the Institute for System Software, as well as my colleagues from the Heuristic and Evolutionary Algorithms Laboratory for interesting discussions, and exchanges.

A huge thank you to the Genetic Improvement community, and the people organizing the GI workshops. It was always fun to attend, and you gave me excellent feedback. Shin Yoo readily became the second reviewer of this thesis, for which I am grateful. I had a great research stay at the University College London - Centre for Research on Evolution, Search and Testing (CREST). Thank you all. Bill Langdon, not only made this possible, and also often collaborated with me since. The same goes for Justyna Petke. I learned a lot from you two!

Also, a thank you to the compiler research community. Christophe Dubach organized a stay at the University of Edinburgh, enabling many interesting discussions with the LIFT research group.

Emmanuel Helm and Andreas Schuler both spent many hours in and out of the office discussing ideas, giving feedback, collaborating on papers, and sometimes just venting over drinks. Thank you! The same goes for the other members of my research group AIST, in no particular order, Martin, Herwig, Lisi, Rainer, Konstantin, Andreas, Christoph, Simone, Michael, David, Eva, Daniel, Barabara, Gerald, and all the other members that have been with us over the years.

Thank you to the people in my life that probably made sure I didn't go insane during this work. My friends John, Philipp, Sebastian, Vedran, Kevin, Lena, Alex, Reinhard, and Christian, we had fun all over the world. Lexx, Max, Georg, Flo, Alex, Armin, Andrea, Phil, Hansi, Maja and Emmanuel for great games and excellent food.

My greatest thanks goes to my mother, Johanna, who has supported my academic career, and always encouraged reading and learning in me.

I also want to thank you. I didn't forget about you, I just knew you were shy, hence I didn't list your name. To all of you who reading the rest of this thesis, thank you too. I'm always happy to answer questions, or to collaborate if you find this work interesting.

Danksagung

Diese Dissertation dauerte beinahe sechs Jahre. In diesen habe ich viel über wissenschaftliches Arbeiten und einige Forschungsfelder gelernt. Es gibt allerdings immer noch sehr viel zu lernen. Viele Personen haben mich in dieser Zeit unterstützt.

Hanspeter Mössenböck ist ein hervorragender Betreuer, der immer zügig und detailliertes Feedback gibt. Ebenso durfte ich von Michael Affenzellers Expertise profitieren. Ich danke euch beiden für die Unterstützung! Danke auch an das Institut für Systemsoftware, und meine Kollegen vom Heuristic and Evolutionary Algorithms Laboratory für den Austausch und die Diskussionen.

Vielen Dank an die Genetic Improvement Gemeinschaft und den Organisatoren der GI Workshops. Es war immer spannend teilzunehmen, und das Feedback war wichtig für den Dissertationsverlauf. Danke an Shin Yoo, für die Bereitschaft der Zweitgutachter der Arbeit zu werden. Ich hatte einen lehrreichen Forschungsaufenthalt am University College London - Cente for Research on Evolution, Search and Testing (CREST). Dieser wurde durch Bill Langdon organisiert, der, ebenso wie Justyna Petke, mit mir kollaboriert. Ich habe von euch beiden viel gelernt!

Auch ein Dankeschön an die Community um den Compilerbau. Christophe Dubach hat einen Forschungsaufenthalt an der Universität Edinburgh organisiert, und einen interessanten Austausch mit der LIFT Forschungsgruppe organisiert.

Meine Kollegen Emmanuel Helm und Andreas Schuler haben viele Stunden mit mir Ideen diskutiert, Feedback gegeben und gemeinsam Publiziert. Danke euch beiden, und unserer Forschungsgruppe AIST. In keiner bestimmten Reihenfolge, Martin, Herwig, Lisi, Rainer, Konstantin, Andreas, Christoph, Simone, Michael, David, Eva, Daniel, Barabara, Gerald und den anderen Mitgliedern, die uns über die Jahre begleitet haben.

Danke an meine Freunde, die sichergestellt haben, dass mich diese Arbeit nicht in den Wahnsinn treibt. John, Philipp, Sebastian, Vedran, Kevin, Lena, Alex, Reinhard, und Christian, die mit mir mittlerweile fast alle Kontinente bereist haben. Lexx, Max, Georg, Flo, Alex, Armin, Andrea, Phil, Hansi, Maja und Emmanuel für gute Spieleabende und besseres Essen.

Mein größter Dank geht an meine Mutter Johanna, die Lesen und Lernen immer gefördert hat, und mir meine akademische Karriere ermöglicht hat.

Zuletzt geht mein Dank an Dich, ich habe Dich nicht in den Aufzählungen vergessen. Ich weiß, dass Du schüchtern bist, und damit ist Dein Name nicht gelistet. Danke, dass ihr die Arbeit lest. Falls ihr Fragen habt oder zusammenarbeiten wollt bin ich gerne verfügbar.

Abstract

Writing source code is a challenging task, requiring the understanding of complex concepts, algorithms and programming paradigms. This task becomes increasingly challenging when source code has to be optimized for non-functional properties such as run-time performance, memory usage or energy efficiency. These properties often depend on in-depth knowledge of the language, the compiler and even the hardware architecture the source code will be run on.

This thesis deals with the identification and verification of patterns in software that influence such non-functional properties. The goal is to help get an understanding, which patterns should be considered or avoided when optimizing towards a certain feature, and where to apply these patterns in the software. This is achieved by the novel combination of two distinct fields of research. *Software Pattern Mining* and *Genetic Improvement*, a Search-Based Software Engineering technique. The approach is applied directly at the compiler and interpreter level. This enables mining of a fine-granular software representation, combining it with in-depth information about the language, and gathering fine-granular performance measurements.

A novel algorithm *Knowledge-guided Genetic Improvement* is presented that allows the generation of multiple semantically equivalent versions of software. These are then mined for discriminative patterns via the novel Independent Growth of Ordered Relationships algorithm. Several bug patterns, often occurring in the mutation operation of Genetic Improvement (GI), have successfully been identified and proven to produce bugs with an average confidence of 90.1%. Fix patterns reduce these bugs with a confidence of 94%. Identified patterns have been successfully applied in the field of *Genetic Improvement* by doubling population diversity, and reducing failing individuals in the population to 36.9% compared to 80% identified in related work. This allows *Knowledge-guided Genetic Improvement* to improve the run-time performance of 22 out of 25 algorithms by an average of 33.5%. The work also successfully identifies anti-patterns and patterns in the run-time domain that are responsible for this improvement.

This thesis lays a foundation for identifying and verifying patterns in the non-functional domain. As a side effect, it also makes the results of experiments in the domain of Genetic Improvement explainable. This opens up opportunities to drive research in the domains *Software Pattern Mining* and *Genetic Improvement* even further. In the future, identified patterns may even be directly applicable in a compiler or interpreter.

Kurzfassung

Das Entwickeln von Software ist eine anspruchsvolle Aufgabe, die ein Verständnis von komplexen Konzepten, Algorithmen und Programmierparadigmen erfordert. Diese Aufgabe wird noch schwieriger, wenn Software für bestimmte nicht-funktionale Anforderungen, wie Laufzeit, Speichernutzung oder Energieeffizienz optimiert werden soll. Dieses Verhalten ist oft von der Sprache, dem Compiler und auch der Hardwarearchitektur abhängig, auf der die Software ausgeführt wird und erfordert fundierte Kenntnisse über diese.

Diese Arbeit beschäftigt sich mit der Identifikation und Verifikation von Patterns in Software, die solche nicht-funktionale Anforderungen beeinflussen. Das Ziel ist es, verständlich zu machen, welche Patterns, an welcher Stelle in der Software, bei der Optimierung einer bestimmten nicht-funktionalen Anforderungen angewendet oder vermieden werden sollen. Dies wird durch die neuartige Kombination zweier Forschungsfelder erreicht. *Software Pattern Mining* und *Genetic Improvement*, eine Methode der Suchbasierten Softwareentwicklung. Diese Methodik wird direkt im Compiler und Interpreter angewendet. Dadurch wird die Suche fein-granularen Repräsentationsformen von Software ermöglicht und mit detaillierten Informationen über die Sprache sowie fein-granularer Laufzeitmessungen ermöglicht.

Ein neuartiger Algorithmus *Knowledge-guided Genetic Improvement* wird vorgestellt, der die Generierung mehrerer semantisch äquivalenter Versionen von Algorithmen ermöglicht. Diese werden mit dem *Independent Growth of Ordered Relationships* Algorithmus nach diskriminativen Patterns durchsucht. Identifizierte Bug-Patterns, die oft im Mutations-Operator von *Genetic Improvement* entstehen, werden mit einer durchschnittlichen Sicherheit von 90.1% bestätigt. Patterns zur Prävention der Bugs werden mit 94% Sicherheit bestätigt. Diese Patterns wurden erfolgreich in der Domäne *Genetic Improvement* angewendet. Die Populationsdiversität wird verdoppelt, und Individuen, die Laufzeitfehler erzeugen, werden auf 36.9% im Vergleich zu 80% aus verwandten Arbeiten reduziert. Dies erlaubt Knowledge-guided Genetic Improvement die Laufzeit von 22 aus 25 getesteten Algorithmen im Durchschnitt um 33.5% zu reduzieren. In Folge werden Anti-Patterns und Patterns identifiziert, die für diese Verbesserung verantwortlich sind.

Diese Arbeit legt eine Grundlage für die Identifikation und Verifikation von Patterns für nichtfunktionale Anforderungen. Als Nebeneffekt können so die Ergebnisse von Genetic Improvement erklärbar und nachvollziehbar gemacht werden. Dies eröffnet Möglichkeiten diese Forschung weiter zu vertiefen und voranzutreiben. In Zukunft können möglicherweise sogar Muster identifiziert werden, die direkt im Compiler oder Interpreter anwendbar sind.

Contents

Pattern Mining and Genetic Improvement in Compilers and Interpreters

Introduction 1

Non-Functional Properties (NFPs), bugs and functional issues, such as security- or maintainability issues, and faults in a compiler or interpreter can all be identified in the source code of software. This work deals with *the identification and verification of patterns* in source code to help find recurring patterns responsible for bugs, and gain an understanding of patterns concerning their influence on NFPs.

Writing source code is a challenging task, requiring an understanding of command structures, mathematics, algorithms, and high-level concepts such as programming paradigms. This task becomes even more challenging when source code has to be optimized towards specific NFPs, such as run-time performance, memory usage or energy efficiency. This requires not only an understanding of the compiler and the virtual machine in use, but possibly even the hardware architecture that the source code will be run on [1]. Automatically identifying and verifying recurring patterns, as presented in this work, can help developers gain an understanding of what impact source code has on NFP.

The identification of patterns that impact software quality and bugs has led to a variety of inspection tools in integrated development environments and external tools. Work in this area deals with the identification of locations, patterns and often suggestions for fixes [2, 3].

The development of compilers, and code optimizations is a challenging task as well. Modern programming languages need to support multiple hardware architectures, automated optimization of code and utility features such as garbage collection. Additionally, standard libraries and handling of low-level tasks such as networking, file access, etc. have become an expected standard. This makes the development of new languages a challenging undertaking [4]. Maintaining the correctness of source code during compiler optimizations and ensuring that no security vulnerabilities are introduced is also a challenging task [5].

The identification of patterns in this work is done directly at the level of the compiler or interpreter in the form of Abstract Syntax Trees (ASTs), and a method to validate these patterns. The advantages of this approach are the general applicability of identified patterns, as the results can be directly integrated in said compiler or interpreter. In addition, this enables direct access to feature not available to similar pattern identification methods in the source code, such as access to the stack and heap and knowledge of the intermediate representations used during compilation and execution.

This work also deals with the domain of *Genetic Improvement (GI) in compilers and interpreters*, in two different capacities. On the one hand, GI can be used as a tool to produce multiple versions of source code without manual overhead, enabling the evaluation of non-functional properties and mining of recurring patterns according to these properties. On the other hand, GI can benefit from pattern mining itself, as the search space can be tuned towards patterns, and patterns known to produce bugs or unviable individuals can be excluded.

1.1 Challenges

The goal of this work is *the identification of patterns in source code* to help developers understand how their code impacts NFPs, and how it may be improved. I.e. the identification of patterns that are semantically equivalent but have different NFPs.The identification of patterns often responsible for bugs is also a goal.

To achieve this goal, the following research questions have to be answered:

RQ1. *How can recurring patterns be identified that impact or improve a functional or Non-Functional Property?*
This question relates to identifying patterns with distinctive impacts, e.g. patterns with a negative or positive impact on run-time performance. Patterns identified in a similar domain can then be used to identify the location of a negatively impacting pattern and suggest replacing it with a semantically equivalent pattern that has a positive impact.

RQ2. *How can the confidence in patterns be improved?*
Without verification an optimization can't be guaranteed to be always correct, and even then the context a pattern is embedded in, or simply the differences in the hardware architecture or the virtual machine executing the code may change the impact a pattern has or may lead to unintended semantic changes. The confidence that a pattern being replaced actually improves the target NFP, while maintaining semantic equivalence, are both important.

RQ3. *How can these patterns be utilized to lead to general optimizations?*
The identification of patterns, and to make them understandable for developers, is a significant first step. The logical next step is to attempt an application of these patterns to improve source code.

These research questions, and the way chosen to answer them, lead to several challenges that need to be overcome. Solving these challenges in addition to the research questions has led to novel contributions in the domains of GI and pattern mining:

Finding *relevant* patterns for specific NFPs The challenge herein lies in the accurate measurement of a given NFP and reaching an acceptable quantity of observations. Modern compilers have a warm-up time in which they analyze code and then perform optimizations most promising in the context of the observed execution behavior [6]. Additionally, side effects from the operating system and other processes may negatively impact the quality of measurements taken. A larger quantity of code with similar functional-goals but different syntax is also needed to enable identification of relevant patterns and to minimize the risk of missing counterexamples that would disprove a pattern to have an impact on a NFP.

Large search spaces Programming languages have a multitude of supported mathematical operators (+, -, mod, ...) branching constructs (if, loops, ...) and concepts [7]. These pose a challenge in GI which usually deals with a manual restriction of the operators and operands used in experiments [8], which is not applicable to a generalized optimization technique in a compiler or interpreter. Similarly, the mining of patterns has to deal with a more intricate and fine-granular search space than comparable approaches, as the source code is considered at a more detailed level.

Validation of patterns reliably verifying that a source code change has a desired impact is challenging. A pattern may only be applicable in a specific context, such as an algorithmic domain, and not generalizable.

Unviable solution candidates in GI . A large part of candidates generated by GI fails to compile, or alternatively produce run-time exceptions during execution [9, 10]. Similar to the search space, this is in part reduced by manual selection of operators and operands, as well as which parts of the code will be modified.

1.2 Contributions

This thesis makes several contributions to the state-of-the-art. These are summarized by their respective areas of research, i.e. in pattern mining and GI. This is provided open source to ensure reproducibility of this work and to make a technical contribution as well.

Contributions in the domain of pattern mining are:

Independent Growth of Ordered Relationships (IGOR) is a novel discriminative pattern mining algorithm based on the concept of pattern growth. It is highly efficient due to only evaluating significant patterns, and extends discriminative pattern mining beyond two groups to an arbitrary amount of groups that can be compared at the same time with multiple metrics (see Section 4.6).

Pattern evaluation due to the combination with GI, confidence in patterns can be strengthened by applying the patterns in experiments, i.e. verifying the assertions about a NFP by creating mutants around the pattern or checking if a bug still occurs when an anti-pattern is excluded from the search space (see Section 4.7).

Programming language specific pattern mining is an extension of the state-of-the-art pattern mining in source code. It groups the constructs of a language into a taxonomy and allows the mining of patterns at any layer of the taxonomy. This enables a generalization mechanism of patterns, and makes the mining approach applicable on different levels of granularity (see Section 4.2).

Bit-based encoding is a novel encoding for ASTs that separates the structure of an AST and the content of the nodes. The encoding mechanism aims for a minimal memory-footprint of encoded trees and enables efficient comparison operations between ASTs (see Section 4.5).

Contributions in the domain of Genetic Improvement and Search-Based Software Engineering (SBSE) are:

Knowledge-guided Genetic Improvement (KGGI) is an algorithm that can apply mined patterns in GI. It gains knowledge by an in-depth analysis of a programming language, and utilizes this knowledge in a dynamic operator graph to restrict the search space to viable candidates (see Section 3.4). KGGI uses patterns by excluding anti-patterns from the search space or by attempting to modify ASTs by inserting positive patterns.

Semantic verification in Genetic Improvement is an open challenge in the field of GI usually solved with test-based verification [11]. This work deepens the existing test-based approach in this area by also considering the test coverage, and ranking tests by their complexity and overlap of functionality tested (see Section 3.1 and Section 3.2).

Sequential and parallel Genetic Improvement algorithms using the complexity of a test suite as a new type of fitness function and two algorithms utilizing this function are introduced. The algorithms work by splitting the suite according to observed metrics and increasing the selection pressure over multiple iterations (see Section 3.3).

Technical Contributions are:

MiniC is a subset of ANSI C 11 and was developed to evaluate and showcase this work with a small language (see Chapter 9)

Amaru is a framework on top of the Graal compiler and the Truffle interpreter. It integrates GI with Truffle and also provides access to external heuristic tools. It also serves as a tool for mining source code for patterns. Furthermore, it provides Java implementations for the aforementioned algorithms in the GI and pattern mining domains (see Chapter 10).

HeuristicLab Connector is a connector between the HeuristicLab framework for heuristic and evolutionary algorithms and Amaru, enabling running experiments via HeuristicLab in a distributed environment.

1.3 Chapter Overview

In Chapter 2 the necessary background from the state-of-the-art that this work builds upon is summarized. Genetic Programming (GP) and GI are introduced in Section 2.1. Frequent, significant and discriminative pattern mining and the state-of-the-art AST mining algorithm are discussed in Section 2.2. The Graal compiler and the prototyping framework and interpreter Truffle are discussed in Section 2.3 and Section 2.4. How they work and represent source code has a lot of influence on the approach introduced in this work.

Genetic Improvement in Compilers and Interpreters is discussed in Chapter 3. The chapter deals with generating different versions of a given program to enable analyzing its NFP and to identify patterns in later chapters ^{RQ1}. In Section 3.1 the test-based approach towards preserving semantics is discussed and compared to other approaches for maintaining semantic validity in GI. Section 3.2 expands upon this by considering the complexity and coverage of tests in GI fitness functions. Two novel algorithms, sequential and parallel GI are introduced in Section 3.3

RQ1: How can recurring patterns be identified that impact or improve a functional or Non-Functional Property?

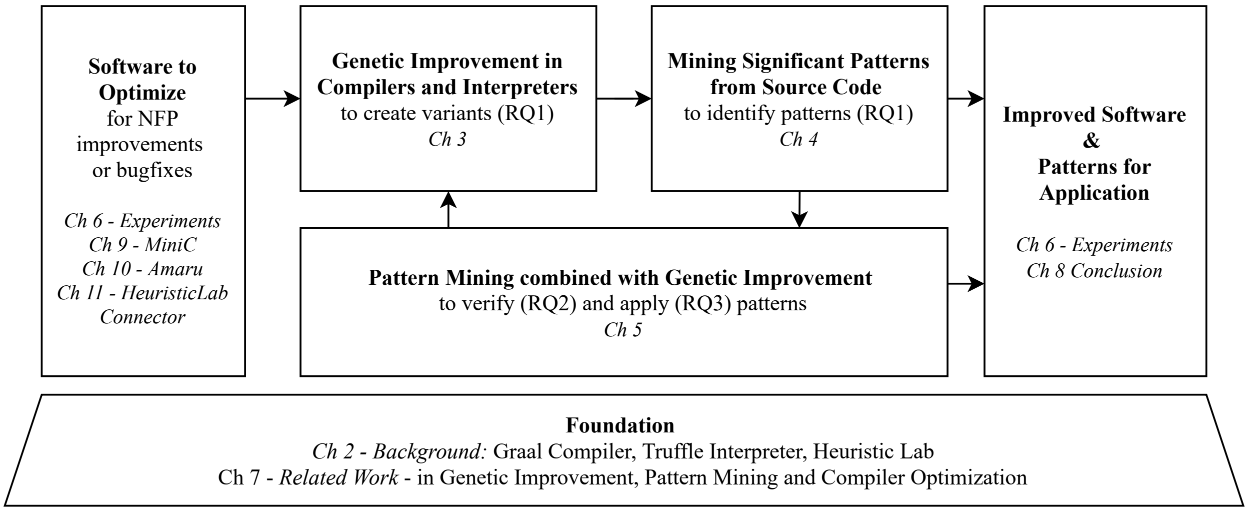

Software to Optimize for NFP improvements or bugfixes
Ch 6 - Experiments
Ch 9 - MiniC
Ch 10 - Amaru
Ch 11 - HeuristicLab Connector
Genetic Improvement in Compilers and Interpreters to create variants (RQ1) *Ch 3*
Mining Significant Patterns from Source Code to identify patterns (RQ1) *Ch 4*
Pattern Mining combined with Genetic Improvement to verify (RQ2) and apply (RQ3) patterns *Ch 5*
Improved Software & Patterns for Application
Ch 6 - Experiments
Ch 8 Conclusion
Foundation
Ch 2 - Background: Graal Compiler, Truffle Interpreter, Heuristic Lab
Ch 7 - *Related Work* - in Genetic Improvement, Pattern Mining and Compiler Optimization

Figure 1.1: Overview of the chapters and their respective topics.

utilizing tests to divide and conquer the complexity of functions to be optimized. Finally, KGGI is introduced in Section 3.4 explaining the syntax graph designed to restrict the search space and the operators utilizing the syntax graph.

Pattern Mining (Chapter 4) deals with identifying patterns that are suspected to be responsible for differences in NFPs^{RQ1}. It discusses the concept of applying taxonomies to source code in Section 4.2 and enriching patterns with wildcards in Section 4.3. A novel encoding presented in Section 4.5 is utilized during the mining process. The expansion of discriminative pattern mining to multiple categories and a developed novel algorithm for mining are presented in Section 4.6. Found patterns can be verified or discarded by utilizing GI as shown in Section 4.7 and outliers can be analyzed further by introducing a multistep mining approach presented in Section 4.7.

Pattern mining combined with GI can verify patterns, as shown in Chapter 5^{RQ2}. Section 5.2 shows how GI can be specifically targeted towards creating trees for pattern mining to find out which parts of the search space should be explored, and patterns that can restrict the search space to not produce run-time exceptions^{RQ3}. This can be done via introducing requirements engineering to the KGGI which is discussed in Section 5.5.

RQ2: How can the confidence in patterns be improved?

RQ3: How can these patterns be utilized to lead to general optimizations?

Chapter 6 contains the empirical evaluation to answer the research questions. The preceding chapters do not contain evaluations, as the relationship between KGGI and pattern mining is cyclic in nature. KGGI is used to generate data that patterns are mined from. In return the pattern mining provides anti-patterns that KGGI avoids in runs.

The related work in Chapter 7 analyzes the domains this work touches. GI operators, algorithms and concepts are discussed in Section 7.2. Approaches and application domains for pattern mining are shown in Section 7.1 and algorithms for this purpose in Section 7.1.

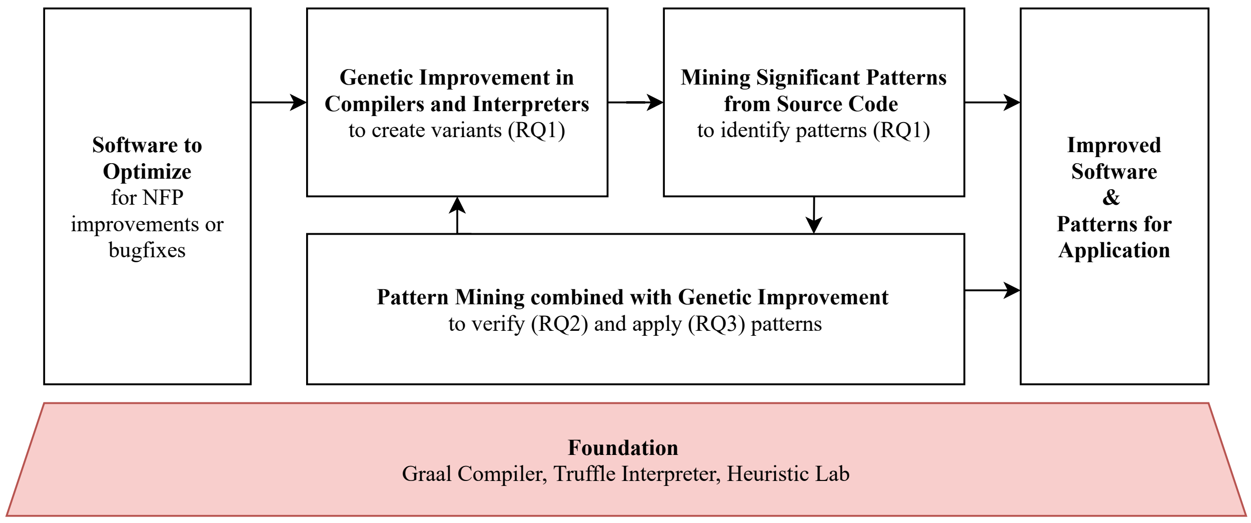
The first part of this thesis ends with the conclusion and outlook of how this work can be improved further, and directions research may take in Chapter 8. The second part of the thesis is continued in Part 8 and summarizes the technical contributions and case studies^{RQ3} for this work.

Background 2

This chapter explains the underlying foundation that this work builds upon. Genetic Programming (GP) and Genetic Improvement (GI) are used in this work to create multiple versions of Abstract Syntax Trees (ASTs) with different functional properties (e.g. bugs, or bug-fixes) or Non-Functional Property (NFP), such as run-time performance. As compilers and interpreters have much larger search spaces than regular GI, the focus is set on the exploration of the search space and the use of test-based methods to evaluate the validity of trees.

Pattern Mining is used to identify recurring structures in the source code that have a positive or negative influence on functional or NFPs. This is based on previous work in discriminative pattern mining and builds up from there. In this work patterns are also applied in GI operators and as independent rewrites for source code.

When considering the application of both of these fields in the context of a compiler or interpreter, a lot is depending on them. Different compilers will follow different optimization strategies, which will influence how the GI approach has to work. One example is code motion, i.e. moving the order of statements or parts of a boolean expression, which is also used in GI operators [12]. It is also a compiler optimization [13]. In compilers that already apply the optimization the GI mutation will have little or no effect, but in compilers that do not, it can still be useful. Thus, the *Graal* compiler and the *Truffle* interpreter are discussed, as they build the foundation of how GI and pattern mining are applied in compilers and interpreters in our work.


Software to Optimize
for NFP improvements or bugfixes
Genetic Improvement in Compilers and Interpreters
to create variants (RQ1)
Mining Significant Patterns from Source Code
to identify patterns (RQ1)
Improved Software & Patterns for Application
Pattern Mining combined with Genetic Improvement
to verify (RQ2) and apply (RQ3) patterns
Foundation
Graal Compiler, Truffle Interpreter, Heuristic Lab

2.1 Genetic Programming and Genetic Improvement

GP can be applied as a Search-Based Software Engineering (SBSE) technique, specifically based on genetic algorithms. It deals with the creation of source code via an evolutionary process to serve a specific task [14]. It is applied successfully in a wide variety of areas such as symbolic regression, prediction and compression [15], even going as far as producing models and code that is considered competitive to code written by humans [16].

GI can be seen as a specialization of GP. Instead of broadly dealing with the generation of code, it deals only with the improvement of existing code. It has been successfully applied to improve run-time performance [10], even genetic programming itself [17] and fixing bugs in software [18–20]. In addition to dealing with existing code, GI also usually deals with smaller parts of the source code compared to GP. This often concerns bug locations, and the types of changes conducted, which often are moving, copying or deleting single lines [12, 21].

GI as well as GP are based on genetic algorithms, and thus have the main operators *selection*, *crossover* and *mutation*. The process is shown in Figure 2.1, beginning with a configuration of the algorithm, such as the selected operators, termination criteria and size limits of individuals and populations. The genetic approach creates an initial population of multiple individuals via the *create* operator. Individuals are evaluated and assigned a quality score via a *fitness function*. These individuals have a *genome*, i.e. a representation form that can be recombined. This population is then evolved by breeding in which two or more individuals are *select*ed usually based on their quality calculated from the *fitness function*, and used in a *crossover* producing offspring that contains parts of both of their genes. Randomly a *mutation* happens to that offspring as well. *Elitism* is sometimes applied in genetic algorithms to keep the best n individuals and to transfer these individuals to the next generation without changes. Elites and offspring from mutation and crossover are then part of a new generation, which in turn continues the breeding process. This process continues until a stopping condition is met, such as reaching enough generations, no more diversity in the population, or reaching a quality threshold [14, 15, 17].

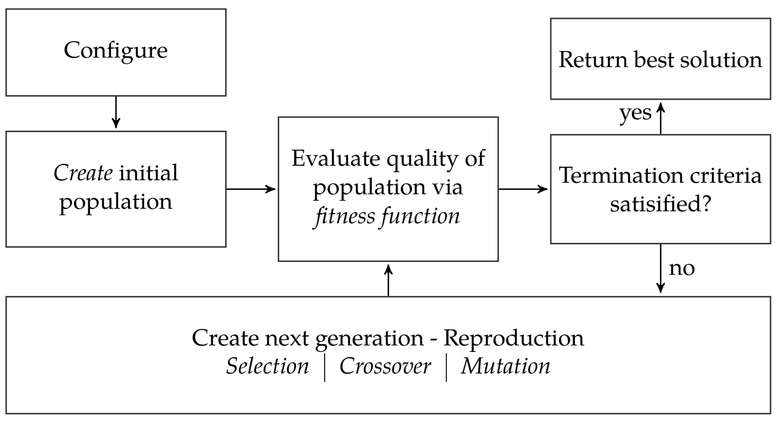

Configure
Create initial population
Evaluate quality of population via *fitness function*
Termination criteria satisified?
yes
Return best solution
no
Create next generation - Reproduction
Selection | *Crossover* | *Mutation*

Figure 2.1: The basic GP algorithm as defined in [14]. Via an initial *create*d population of individuals, multiple generations are bred via *selection*, *crossover* and *mutation*, and evaluated via a *fitness function*.

The *create* operation in GI, as it improves existing software, usually starts out with mutants of the original, and is similar or equal to the *mutate* operation. Both, *create* and *mutate*, often conduct *grafting* [22]. Grafting is the process of taking existing source code from somewhere else, and integrating it into an individual in the population. The graft often comes from other functions in the same program. This process works well as the same code base often works with similar concerns. For bug fixes, some locations of a larger code base may already have a fix that was not applied at all locations the bug might occur [23, 24].

Crossover operations are dealt with in a wide area of approaches, that often also influence all other operations. For example *homologous crossover*, i.e. crossover only at positions that are similar in type or are in a similar context, has been proposed for genetic programming [25]. Genetic Improvement goes even further with *Fertile Darwinian Bytecode Harvester (FINCH)*, which conducts a crossover while maintaining control flow and validity of local variables [26]. *Grammar Guided Genetic Programming (GGGP)* works by using a grammar for the creation of code, reducing compilation errors by ensuring that the produced code follows the given grammar [27, 28].

The *fitness evaluation* in GI is done via test-based evaluation [12, 29] or in some cases via formal software specification measures [29]. The test cases are usually part of a test suite, where the fitness is calculated via the number of passed and failed test cases. If all tests pass, the individuals are considered as semantically correct, i.e., the bug has been fixed or the functionality has been maintained when improving a NFP. Halting a program that would otherwise never finish is usually addressed by introducing a time limit or restricting the amount of executions via instrumentation [26].

Our work builds upon the very same basis of GI and also applies concepts of maintaining control flow [26] and utilizing the grammar of a language [27, 28]. However, it expands upon these concepts by introducing requirements derived from mined patterns of the individuals produced by the GI process, and improves GI itself by adhering to found good patterns and by preventing anti-patterns. This makes the larger than usual search spaces of compilers and interpreters manageable.

2.2 Pattern Mining

This section discusses the background of *frequent*, *significant* and *discriminative* pattern mining in source code. *Frequent* pattern mining works on the assumption that in a given set of information, certain subsets will be recurring more frequently than others. This information can be used in different domains for mining [30]. It is also often called *significant* pattern mining in a similar context, though frequent mining relates more often to frequently occurring substructures, while *significant* generally is used in conjunction with a metric that decides if a frequent pattern is also relevant. This metric is the *minimum support metric* which is based on how many occurrences a pattern has over the entire data set being analyzed. This essentially introduces a ranking and filtering mechanism for frequent substructures [31–36].

Discriminative pattern mining expands the concept of *frequent* pattern mining by separating the data set into two groups [2, 35, 37–39]. Discriminative pattern mining is usually done in the domain of software fault localization or fault mining, and is the reason why the search space is usually split into the two groups of *failing* and *succeeding*. Thoma et al. [35] are an exception to this. They use a classification approach combined with discriminative pattern mining to identify relevant graphs rather than just discriminative ones, without a specific domain.

Pattern Mining approaches in source code are either done *statically*, meaning that the source code is mined but not executed, *dynamically* meaning that only execution results of the code are analyzed or in a *hybrid* mode.

Static approaches [31, 34, 40–43] primarily have the advantage of the code not having to be executed. This saves the approach the need to make the code executable, and saves time to execute it. Static approaches often analyze snippets from code patches [43, 44]. As a disadvantage non-functional properties can either not be recorded at all, or can only be inferred, for example, from commit messages.

Dynamic approaches [2, 37, 38, 45] have the advantage of being able to use the execution results. This enables the accurate verification of functional properties such as test case executions or call-traces to the executed functions, which are not always identifiable through static analysis. NFPs (e.g. run-time) could also be measured, but this is usually not done. The disadvantage is the overhead that the execution produces before mining can begin.

Hybrid approaches [46, 47] combine the advantages and disadvantages of both approaches. They produce promising patterns for functional and non-functional domains, but are costly to conduct.

The domains that these mining approaches occur in are varied, and contain localization of bugs [2, 39, 45], recommendations for improvement, such as code quality for reviews [43], automating recommendations [41, 48] identification of crosscutting concerns [42] or code clones [40]. Often this work goes on to identify patterns either for bugs [37, 46, 49] or how code is used [31, 34, 44, 47, 50, 51].

All approaches use one of three representation forms. Sequences [34, 42, 43, 46], trees [2, 41, 49] or graphs [31, 37–40, 44, 45, 47, 48, 50, 51]. Sequences most often represent the source code [42, 43], or logs of its execution [34, 46]. Trees often represent an AST [41, 49] or the call tree [2]. Graphs most often represent the source code with additional information such as control or data flow [39, 40, 44, 47], or relationships between them [48, 50, 51]. Otherwise, they represent the call graph [31, 37, 38, 45]

Algorithms in pattern mining are well researched [30, 36, 39, 44, 47, 52, 53] and can be categorized into three different types.

Apriori algorithms [32] start out with all permutations of size 1 in the search space, and continuously creates all permutations of the size n+1 until a stopping condition has been applied or the search space has been exhausted. Pruning happens only after evaluation of each iteration, leading to the creation of many patterns that are unnecessary and are filtered later by pruning metrics.

Pattern growth algorithms such as SLEUTH [30] instead start out with all size 1 patterns, and locations thereof in the search space. Patterns are grown only if a given metric shows that the growth will be relevant in the mining context. The growth usually has optimality guarantees, ensuring that the same pattern is not evaluated multiple times. It also only grows patterns that are relevant, as opposed to *apriori* algorithms that need to grow all patterns since locations are not tracked. Thus, pattern growth algorithms are more efficient than apriori algorithms.

Non-optimality guaranteeing methods were developed for search spaces that are too large to be fully explored. They often do not guarantee that all frequent patterns will be found, but rather use heuristics or clustering [53] to move in a good subset of the search space, often in combination with regular algorithms in the domain [41].

The foundation for this work lies primarily in the concepts of the discriminative pattern mining approaches and algorithms. However, this work expands the concept of discriminative pattern mining even further and tackles the issues identified by Thoma et al. [35], namely redundancy of frequent substructures, and that statistically frequent patterns do not necessarily imply these patterns to be relevant. While Henderson and Podgurski [39] tackle this with classification combined with discriminative pattern mining, our work utilizes the inherent qualities of source code in its representation and natural hierarchies of code concepts. Our work also expands the state-of-the-art concerning the representation form, and introduces a multi-layered mining process. To achieve this, we introduce the novel algorithm Independent Growth of Ordered Relationships (IGOR) instead of using a known algorithm from the state-of-the-art.

The analysis of NFP is entirely absent from the field of pattern mining, except for properties of the code itself such as maintainability or complexity [43]. Features such as run-time performance or energy-performance are not researched. This is another area our work discusses.

2.3 Graal

Graal is a compiler that is a part of the OpenJDK project since JDK 9 [54]. It is an aggressively optimizing just-in-time (JIT) compiler, which translates the bytecode into an intermediate representation (IR), and from there into machine code that is executed. Graal directly integrates with the Truffle framework (Section 2.4), and compiles languages implemented in Truffle. It contains many feedback-directed optimizations, such as inlining, loop unrolling and partial escape analysis for compilation [55–57]. Among those optimizations some are speculative, such as the performance estimation of its own optimizations to choose which optimization to apply [58]. In rare cases, speculative optimizations have to be undone at run time if the assumptions on which they relied turn out not to hold.

The amount of optimizations and the complexity of Graal make it challenging to use Graal in the context of GI. One reason for this is that Graal has a warm-up phase in which it analyzes the behavior of the code to drive the speculative optimizations [57–59]. Also, due to the feed-back-directed optimizations the same AST may have different behavior on multiple compilations, or may behave differently depending on what input is given, e.g. in the context of GI if the test case suite drives a hot path. This makes the evaluation of NFP challenging and time-consuming as the warm-up has to be passed. Graal is so highly optimized, that it is hard to find additional run-time performance optimizations.

The Graal IR is a directed graph made up of both the control flow and the data flow of a given function. This concept is similar to the program dependence graph (PDG) known from the domain of pattern mining [44]. The primary difference is that the Graal IR views the source code at a lower level than the PDG. While this level of granularity is not the focus of this work, exploring the Graal IR for patterns and optimization rewrites may be an interesting future research topic.

2.4 Truffle

Truffle is an interpreter, and framework for implementing and prototyping programming languages. The representation of source code in Truffle is based on ASTs. Truffle is written in Java and can execute on any Java VM. However, it directly integrates with Graal and leverages high performance compilation that is only available on a Graal VM. Truffle itself also has several optimization techniques that it provides for its interpretation, such as node specialization, and loop optimization [60–62].

One of the major advantage of Truffle languages is that they automatically come with the optimizations that the framework offers, and that Truffle is executed in the JVM with features such as garbage collection. Truffle provides several such languages, including JavaScript, C, Python and Ruby. These languages are often open source, and can even interact with each other. For example a JavaScript Truffle node can produce a call to a Python Truffle node [62].

Every node in the AST represents a concept of the language, e.g. "write to stack", "for each" "+". These concepts must be implemented by hand, following the tools Truffle provides as a framework. Listing 2.1 shows example implementations of a stack-read and an integer addition. When looking at the class *MinicIntReadNode*, the access to the VirtualFrame is shown. This is an access to the stack, and this frame provides run-time information on which variables are available on the stack, if they have been initialized, and what their value is. This frame is associated with a frame descriptor providing information without an execution. A similar concept exists for the heap in the form of a MaterializedFrame. This is valuable information that can be provided for GI in its operators to conduct data-flow-sensitive operations, as well as the pattern mining which can identify the variables in the patterns.

The IntArithmeticNode shows how nodes are related. The manual implementation has the @NodeChild annotation, which maps to the parameters of the function add, and in the implementation generated by Truffle, MinicIntAddNodeGen, refers to other nodes in the language. MinicIntAddNodeGen also partially shows the execution in executeInt, and the execute and specialize operation. This operation conducts one of the

Truffle optimizations. The Truffle nodes have specializations based on the data types via the @Specialization annotation. Depending on which data types are fed to the function, nodes can be specialized towards one data type, and thus speed up the execution.

Listing 2.1: Sample implementations for the concepts *read integer from stack* - MinicIntReadNode, and *int addition* - MinicIntAddNode from the MiniC language

```java
@NodeInfo(shortName = "read-local-int", description =
    "Reads int from stack")
@NodeField(name = "slot", type = FrameSlot.class)
public abstract static class MinicIntReadNode extends
    MinicIntNode {
    protected abstract FrameSlot getSlot();

    @Specialization
    protected int readInt(VirtualFrame frame) {
        return MinicFrameUtil.getInt(frame, getSlot());
    }
}

@NodeInfo(shortName = "arith-int", description = "Abstract
    base class for arithmetic int operations")
@NodeChildren({@NodeChild("leftNode"),@NodeChild("rightNode")})
public abstract class MinicIntArithmeticNode extends
    MinicIntNode {

    @NodeInfo(shortName = "+", description = "int + int")
    public abstract static class MinicIntAddNode extends
    MinicIntArithmeticNode {
        @Specialization
        public int add(int left, int right) {
            return left + right;
        }
    }
    ...
}

@GeneratedBy(MinicIntArithmeticNode.class)
public final class MinicIntArithmeticNodeFactory {
    @GeneratedBy(MinicIntAddNode.class)
    public static final class MinicIntAddNodeGen extends
    MinicIntAddNode {
        @Child private MinicIntNode leftNode_;
        @Child private MinicIntNode rightNode_;
        ...

        @Override
        public int executeInt(VirtualFrame frameValue) {
            ...
            int leftNodeValue_ =
    this.leftNode_.executeInt(frameValue);
            int rightNodeValue_ =
    this.rightNode_.executeInt(frameValue);
            return add(leftNodeValue_, rightNodeValue_);
            ...
        }

    private int executeAndSpecialize(Object
leftNodeValue, Object rightNodeValue) {
        ...
    }

    ...
}
```

What is important from how Truffle nodes are built and interact with each other, is the granularity. This type of AST, which represents a function that can be called, is much more fine-granular than other approaches in GI as well as in pattern mining. It also is automatically executable, and the framework itself enables swapping nodes with optimized ones.

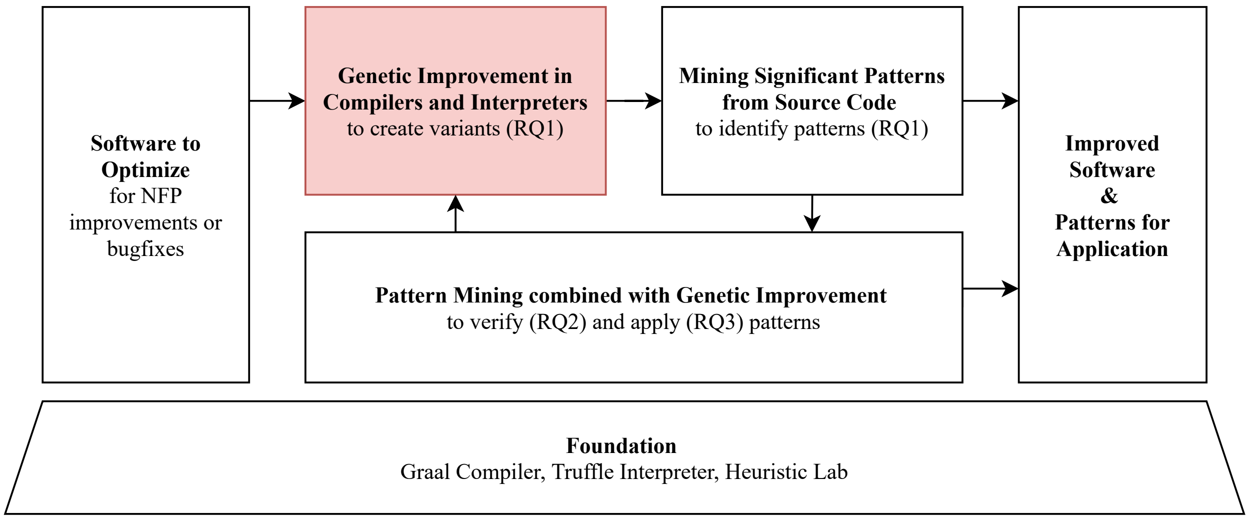
How the nodes are related to each other, produces a natural hierarchy represented by the class hierarchy the nodes are implemented in. For example, an IntAddNode is an IntArithmeticNode, which in turn is an IntNode, which is an ExpressionNode (a node returning a value). In the other direction an Expression node has the Implementation IntNode, DoubleNode, InvokeNode and so on. This can be leveraged for the pattern mining as well as the GI approach.

Genetic Improvement in Compilers and Interpreters 3

Genetic Improvement (GI) is a Search-Based Software Engineering (SBSE) technique, focusing on fixing bugs (functional properties) and optimizing Non-Functional Properties (NFPs) of existing software. It is an adaption of Genetic Programming (GP) which has a broader context of using genetic algorithms to synthesize source code. Primarily, GP is understood to create new functionality, whereas GI modifies existing functionality [63].

GI derives its success from two primary factors. The first being the much smaller search space compared to GP. GI largely tackles localized bug fixes, often modifying a single line or very few lines in a program. In a similar manner, GI has a reduced set of operators (+, -, *, if, for) and operands (literals, function calls, ...) that the generated source code may contain [26, 63]. The second factor is that only existing source code is tackled. Synthesizing new functionality is harder than fixing a bug in existing source code, or adapting that existing source code to better fit a specific NFP [29]. Contributing to this is the fact that code is often repetitive in the same code base, and fixes can be grafted from other parts of the software [24].

In the context of our work, GI finds its use in providing source code that can be mined for patterns. GI is used to create multiple variants of source code, that are later analyzed for recurring patterns that have an impact on NFP or on exceptions that occurred. Pearson et al. [64] shows that this may impact the quality of the pattern mining approach, as artificially introduced faults are vastly different from real faults and thus mining one can't predict the other. However, in the context of compilers and interpreters which need to modify code in a generic and automated way as opposed to an individual and manual change for observed code, this should not be as much of an issue, compared to mining patterns for manual fixes.


Software to Optimize
for NFP improvements or bugfixes
Genetic Improvement in Compilers and Interpreters
to create variants (RQ1)
Mining Significant Patterns from Source Code
to identify patterns (RQ1)
Improved Software & Patterns for Application
Pattern Mining combined with Genetic Improvement
to verify (RQ2) and apply (RQ3) patterns
Foundation
Graal Compiler, Truffle Interpreter, Heuristic Lab

This chapter deals with the specifics of utilizing GI in the context of a compiler or interpreter. This comes with several challenges that GI usually does not have to deal with, or can deal with in ways that are not an option in this context:

Semantic validity is handled in GI almost exclusively via test-based validity [29]. The research in this area focuses on automated test input generation, using the original source code as a test oracle, and optimizing test suites for coverage. Our work does not go beyond these approaches, but does attempt to improve the confidence in test-based verification.

Large search spaces are usually managed in GI by tackling smaller parts of the source code, and manually adapting the used operators and operands [26, 63]. The issue herein lies with the genericity of the approach. In the context of a compiler or interpreter, manual adaptions to the search space are infeasible. Similarly, mining patterns from such manually influenced search spaces would not lead to relevant patterns, but would only identify patterns that reflect the manually pruned search space. Thus, ways to deal with the search space have to be identified.

Infeasible individuals in a population are a common occurrence in GI. Up to 80% of individuals [8, 9, 26] generated with GI techniques cannot even compile. A large amount of individuals also produces run-time exceptions. Knowledge-guided Genetic Improvement specifically tackles this issue.

Expensive evaluations are a general issue in GI, as source code needs to be run to be evaluated. This however is aggravated by the use in compilers and interpreters, especially in the context of NFP. Modern compilers have a warm-up phase in which code is analyzed before it is optimized. To get accurate measurements, they have to be performed after this warm-up phase. For example, run-time performance often stabilizes only after 200,000 [65]. This is aggravated by the large amount of individuals that have to be evaluated in genetic algorithms.

Applying GI in a compiler or interpreter also comes with many advantages that can be utilized, which would not be available otherwise. For example, the approach has access to the information, which variables are available on the stack and heap, as well as the functions that were linked in the context. In addition, measuring during the execution has multiple advantages. The approach can be utilized to exactly know what parts of the code were executed via which inputs, and at which points the compiler or interpreter has performed a modification.

3.1 Maintaining Semantic Equivalence

The key challenge in utilizing search-based software engineering methods, such as genetic algorithms, to modify source code is to maintain the semantics of that code during the modification of a NFP. This is less of an issue when attempting bug-fixes, as the code is already proven to be semantically incorrect. In GI there are two options to maintain semantics, test-based verification and code modifications that do not change the semantics of source code.

Operators That Maintain Semantic Equivalence

The *create*, *crossover* and *mutate* operators are the three core operations in Genetic algorithms. In GP these operators can be defined in such a way, that they do not modify the semantics of source code while only modifying the syntax. The major advantage of this approach is, that, as the semantics of the source code are not changed during the operations, the results will always be correct, and can be utilized in optimizations of NFP. However, these operations are very localized and must be manually crafted for each programming language, or alternatively restrict the search space to such a small part of a language that the semantics can't be influenced very far (called simplification) [66–68].

As an example of such a method, McPhee, Ohs, and Hutchison [66] show semantic building blocks in the context of binary AND, OR, NAND and NOR operations. The behavior between these operations can be manually coded in such a way that any operation can be replaced with a semantically equivalent one.

In the context of compilers and interpreters, such methods are mostly irrelevant, as modern compilers perform such optimizations already. For example, superoptimization [69] is an optimization method that modifies the execution order of code, similar to code motion [13]. This can also be done with genetic algorithms. The MoveLine Operation from the *GI in No Time (GIN)* framework is a very similar concept, though it does not check if the line edit is valid [21, 70].

Looking at it from the other direction, GI could profit from existing operations in compilers. The LIFT compiler was developed for a functional, parallel programming language for graphics processing unit (GPU) applications. Its optimizations are based on rewrite-rules that are written by the compiler developers, and guarantee semantic equivalence while rewriting the syntax of code. This has been successfully used in Convolutional Neural Networks (CNNs) to optimize source code [71], and similarly could be applied via GI, at least in the mutation operator.

This way of maintaining semantic equivalence is generally not suited for mining patterns, as the patterns would only identify what the operators are already doing or would identify optimizations already made by the compiler or interpreter. Thus, this method of maintaining semantic validity is not utilized in this work.

Test-Based Verification of Semantics

Test-based validation is widely used in GI, as the unmodified source code can be utilized as a test-oracle, i.e., it can be executed against any desired input to produce test cases on the fly. In addition, automatically fixing bugs in software often means that bug-revealing tests already exist, making it easy to utilize these tests to automate bug fixing [29].

Utilizing tests to verify semantics has the disadvantage, that tests do not provide the same confidence in changes to the source code as formal proof would have [29]. However, even well-tested compiler optimizations that are based on formal proof have recently been shown to introduce security bugs [5, 72]. Arguably, due to the complex nature of software, preservation of semantics can never be guaranteed to be perfect in any circumstance, though when using optimizations in the context of a compiler the confidence in the correctness of an optimization should be fairly high.

GI can be used in this work via a test-based verification approach, as it is used as a tool to find interesting optimization patterns. In any case, that these patterns should be analyzed by experts before being introduced in a compiler or interpreter, thus eliminating the disadvantage of test-based verification.

3.2 Test-Based Genetic Improvement

Using tests to validate the semantic correctness of source code has many advantages if done via a compiler or interpreter. The coverage can be measured with a much finer granularity than usual in this setting. The execution environment also provides access to information that would otherwise not be available, such as the stack and the heap. This enables improving the confidence, compared to line-level coverage, that a given Abstract Syntax Tree (AST) is semantically correct, even after being modified via GI, as even partial statements, such as conditionals with short-circuiting, can be measured for coverage.

Much of what follows has been previously introduced in [73] along with introducing two new algorithms and a novel utilization of tests in a fitness function. Since it's introduction, it has been refined and revised, especially within the area of utilizing information available from the execution environment.

Test Coverage

Depending on the compiler or interpreter being used, *code coverage* of a test case can be done with a finer granularity than would be possible otherwise. Code coverage usually goes to the granularity of branches that are being covered. In some circumstances the coverage can be more refined in an interpreter or compiler. One instance is short circuit evaluation, i.e. the left to right execution of boolean operators, where *and* conditions stop evaluating after the first *false*, and *or* conditions stop evaluating after the first *true*.

For example, in the Truffle interpreter (see Section 2.4), the entire language is implemented as nodes that are executed in an AST. An example AST is shown in Figure 3.1, representing a recursive implementation of the Fibonacci sequence. Nodes in green are executed via the given test case $fib(0)$. This enables a rather fine-granular definition of coverage, which in our work is used for one AST being optimized. Let *visited(test)* be the set of nodes visited in the AST during the test, and let *nodes(AST)* be the set of nodes in the AST:

$$\text{coverage}(\text{AST}, \text{test}) = \frac{|\text{visited}(\text{test})|}{|\text{nodes}(\text{AST})|} \tag{3.1}$$

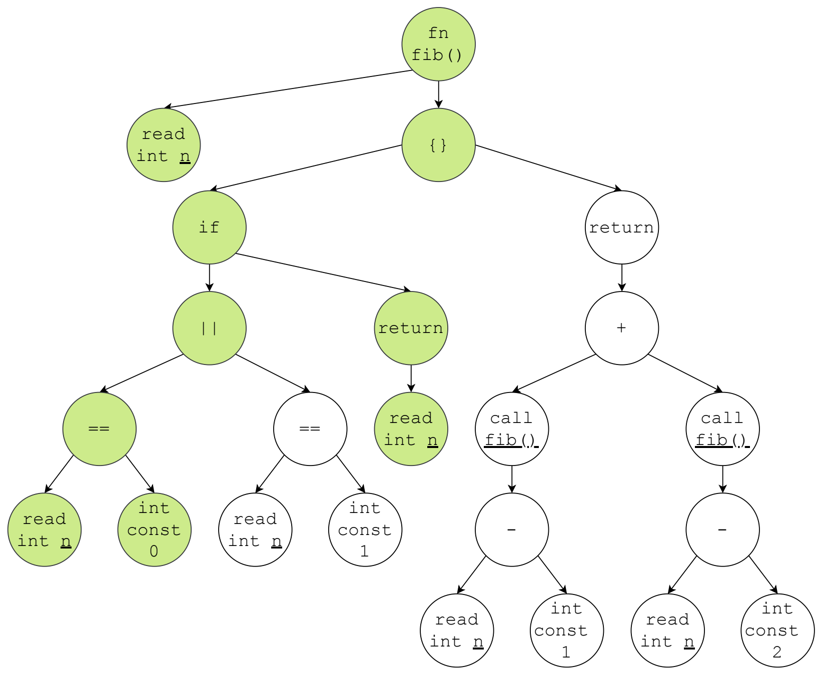


Figure 3.1: AST representation of a recursive implementation of the Fibonacci sequence, nodes with a green background are visited when the input of the function is 0. In this case the second part of the (||) node is not executed due to short-circuiting.

This definition slightly deviates from the well-known test coverage definitions of *statement*, *branch* and *path coverage* [74], but is of the same intent as n% of the code is covered in a test case. To be usable for a fitness evaluation, the coverage of a function is always measured as *node coverage*, via the amount of nodes executed in a given AST. The node coverage also only considers the coverage of the first execution of a function. The reason for this is to avoid direct and indirect recursion, as this often leads to all or most branches being covered, and degrading the discriminative qualities of single test cases, as well as increasing the overlap between them. For example, a recursive call to Fibonacci Figure 3.1 would produce a 100% coverage with just the test input (2) when not excluding recursive calls. This would be desirable for regular testing, but prevents utilizing the node coverage metric in a fitness function (see Section 3.3).

It is important to note that our work assumes that test cases are manually provided or synthesized in a correct way, as generating the tests is beyond the scope of this work. Coverage in this case is used solely to rank tests for test-based GI, not to reduce or optimize the test suite used, which would be inadvisable as research suggests that reducing overlapping test cases does negatively impact the semantic correctness [75]. This also becomes apparent in Figure 3.1, as, using the node coverage metric, two tests with input (1, 2) are enough to produce 100% coverage and success rate of the test suite. The function has an obvious bug, though. If it were to be called with any negative value it would result in an endless recursion, and a stack overflow at run time.

In addition to using the coverage of a single test for ranking that test, the coverage for the entire suite is also used to verify that the entire AST is covered. Let *tests* be the set of tests in the suite:

$$\text{coverageSuite}(\text{tests}, \text{AST}) = \frac{\left|\bigcup_{t \in \text{tests}} \text{visited}(t)\right|}{|\text{nodes}(\text{AST})|} \tag{3.2}$$

This measure is relevant for both the original function that is being optimized, as well as for the optimized AST. In the case of the original function, this is an indicator if the selected test cases are sufficient. In the case of the optimized function, the metric ensures that the AST that was modified has no unintended behavior that is untested.

Evaluating Test Complexity

An additional indicator of tests beyond coverage can be provided by the compiler or interpreter itself, though this feature is one exclusive to the use of *Truffle*. One of the core concepts of the Truffle interpreter is that it has generalized nodes, that can be specialized towards specific data types (for additional information, see Section 2.4). Our approach uses this information as a complexity measure that tests can be ranked by. It can be more generally formulated for any given compiler or interpreter, as stated in Definition 3.2.1.

Definition 3.2.1 *A* ***specialization*** *occurs when any observable modification of the source code happens via the executing or compiling system.*

This specialization metric, among several others, is used in the presented approach to measure the *complexity* of a test case, and is later used in algorithms and fitness functions as an alternative to the coverage measure for ranking and grouping tests:

Specializations , i.e., the amount of modifications the executing or compiling system is performing on those parts of the source code executed during the test case. This measure can help identify test cases and AST that the compiler already optimizes.

Control Structures concern mostly loops, and branching statements that are covered in the test case. The assumption behind this metric is that tests that enter more or different control structures cover different (and more or less complex) functional considerations of the function being analyzed.

Function calls to other functions in the source code represent an optimization boundary in source code. It is likely that a function serves a specific purpose (logging to a database, reading user input, ...) or must be called with exactly same input if the AST is modified by GI. The more function calls there are in the AST, the more complex the optimization becomes.

NFP of any kind can serve to distinguish the complexity of a test case. For example, in the context of optimizing the run-time performance of source code, a valid complexity measure would be the run-time of a single test.

Any of the above complexity measures (defined in Equation 3.3), in addition to coverage, can be considered a valid ranking mechanism for test cases. Which of these mechanisms should be chosen may be different depending on what is being optimized. For example, a purely mathematical function will have no control structures, and thus all test cases will have the same amount of control structures (0) and likely the same amount of specializations towards a specific data type. In other cases, the values will depend on the specific implementation. Considering Figure 3.1 the recursive implementation of Fibonacci has a node coverage of 0.43 (10 out of 23 nodes), and 0 function calls with an input of (0). It has a coverage of 0.92 (21 out of 23 nodes) and 2 function calls when selecting an input of (2) or higher. An iterative implementation may have a similar behavior, but would not have any function calls, thus showing no difference in the function call metric.

$$\text{complexity}(\text{AST}, \text{test}) = \frac{\text{metric}(\text{AST}, \text{tests})}{\max(\text{AST}, \text{tests})} \tag{3.3}$$

Relationships Between Test

In addition to measuring the coverage of a test and its complexity, it is important to understand the relationships between tests on a given AST. This is done via calculating the overlap of given tests, defined as the amount of nodes that two tests (A) and (B) visit compared to the total amount of nodes in a given AST. Let *tests* be the set of tests. Let *visited(A)* and *visited(B)* be the set of visited nodes in tests A and B:

$$\text{overlap}(A, B) = \frac{|\{\text{visisted(A)} \cap \text{visited(B)}\}|}{|\text{tests}|} \tag{3.4}$$

Confidence in Semantic Validity

The previously defined measures are utilized in the approach to rank and distinguish different test cases in a fitness function intended for the use in GI experiments. The metric *confidence* serves the purpose of identifying how likely it is that a solution produced by such a run is semantically correct, i.e., has the same behavior as the original AST when optimizing a NFP.

Algorithm 1: Accuracy calculation of an AST against a given test suite. 0 is perfect accuracy, and a rising value is increasingly more inaccurate.

```
Data: testSuite
/* Initialization                                         */
accuracy ← 0;
/* Iterate through tests                                  */
foreach test ∈ testSuite do
    if test.exception ¬ test.expectedException then
        accuracy ← accuracy + 10;
    else if test.returnType ≠ test.expectedType lor (test.result = null ∧
      test.expectedResult ≠ null) then
        accuracy ← accuracy + 2;
    else
        accuracy ← compare(test.output, test.expectedOutput);
    end
end
Result: accuracy
accuracy ← accuracy / count(testSuite);
```

To define how accurate a test is, the difference between the expected outcome and the real outcome of a given test must be made measurable. In addition, this must be done independently of a specified data type.

One side effect from applying GI in a representation that the interpreter or compiler use, is that the AST being optimized may even return a different data type than the original AST, including null values instead of expected values and vice versa. Also, the AST may throw an exception that was not intended, or not throw an expected exception. Algorithm 1 summarizes how the accuracy is calculated.

The *compare* function in line 8 of Algorithm 1 serves to normalize any given data type difference between 0 and 1. This is done by calculating "|output−expectedOutput|/(double)largestOutput" where "largestOutput" is the largest output of that data type observed over all tests cast to a floating point value to prevent rounding errors. This allows the normalization of any integer or floating point value. String values are compared via computing the Levenshtein distance (the amount of single character edits to transform word a into word b), the "largestOutput" in this case is the length of the largest observed string, which would also be the maximum number of edits possible. A complex data type (struct, object, ...) can be transformed via this compare function by simply using the sum of the compare results over all fields.

The *confidence* is a combined measure of the code coverage, along with the correctness of the code. The coverage is included to verify that no unexpected behavior is also covered in the result. This ensures that additional behavior that was not tested is not contained in the resulting AST. The correctness is a measure that, independent of any data type, allows us to determine how close a given test was to the original result. Depending on what is being optimized, this measure may not necessarily be required to be 0 (i.e. 100% correct). Considering floating point data types, slight deviations are acceptable. Considering the type of algorithm, for example shaders for graphics applications, an AST that produces a higher frame rate than the original but is not as accurate may be an acceptable trade-off. As the coverage goes from 0 to 1 where 1 is best, and the accuracy from 0 to ? where 0 is best, 1 + accuracy is inverted resulting in a perfect score of 1 if it is accurate, and moving closer to 0 the less accurate it is.

$$\begin{aligned}&\text{confidence}(\text{tests}, \text{AST}) = \\ &\qquad \text{coverageSuite}(\text{tests}, \text{AST}) * \frac{1}{1 + \text{accuracy}(\text{suite}, \text{AST})}\end{aligned} \tag{3.5}$$

Expanding confidence beyond coverage

There is a wide area of research in the topic of software testing, ranging from automatic test synthesis [11], quality of test suites [75] and co-evolution of test cases with a GP experiment [76]. These approaches go beyond the scope of this work. They would, however, greatly improve the confidence in the results that GI produced in this approach, and are thus shortly outlined as possible future work. This work assumes that an adequate test-suite is provided for a function being optimized.

Zhu, Hall, and May [74] elaborate on test adequacy or *mutation adequacy*, which is defined by how many mutants of code that have intentionally inserted bugs, would be detected by a given test set. Mutation adequacy in this context is interesting, as it can be integrated directly in the approach. GI naturally produces mutants, that could be used to verify if a given test suite is adequate for the purpose of optimization.

In a similar way, additional test cases could be synthesized directly during the experiment [76], either via existing synthesis methods or directly via a genetic algorithm, and this too could be supported by considering coverage. While using the overall coverage of the entire test suite for the original program as a starting point, the coverage against an AST in the GI population can also be measured. If the coverage of the modified AST is not similar to the original AST, this can either indicate that dead code has been introduced by the algorithm or, alternatively, that new tests should be synthesized as they do not cover the entirety of the program.

3.3 Dynamic Fitness Functions Based on Test Grouping

The application of dynamic fitness functions, is based on the previously introduced measures of *complexity* (Equation 3.3) and *overlap* (Equation 3.4). A fitness function in a genetic algorithm usually does not change over the execution of an experiment. This section introduces two novel algorithms extending the core genetic algorithm approach, which modify their fitness function during an experiment.

The core idea of this concept, originally introduced in [73], is that this modification moves away from a static fitness landscape, towards a dynamic one, with the fitness function continuously increasing the selection pressure towards the target functionality. This can help, as maintaining the semantic correctness of a given AST produces a flat fitness landscape with few optima over an entire test suite, but single selected test cases or subsets of the test suite have a fitness landscape with more optima, allowing the genetic algorithm more leeway to produce valid code in one area.

Depending on how the test case is selected, and grouped or ranked, this indirectly influences the hidden functional or non-functional properties of code. For example, null checking at the beginning of a function may be used as a form of defensive programming. Similarly, the if check of Figure 3.1 serves as the stopping condition for the recursion, but a check for negative values is missing, resulting in a bug. The functional property of this function is clear, "produce the Fibonacci sequence". Defensive code is not among the non-functional properties. And a hidden non-functional property would be the exponential run-time complexity of the function, $O(2^n)$. This would be virtually unnoticeable with smaller numbers, but becomes problematic with higher numbers, and would in a recursive implementation as given also lead to a stack overflow relatively fast. Thus, a hidden functional feature would be the upper limit of numbers that could be given as input.

For this example, ranking the test cases by their measured run-time can serve to control the search space. The core functionality would still be served with the three test cases of a respective input of (0, 1, 2). A second group (1000, 2000, 3000) can serve to measure how the approach scales, as well as if it is resistant to a stack overflow, and would not fail if the data types used in the calculation are large enough. This very simple example already shows that the selected complexity measure has an impact on the fitness landscape. Just wanting to ensure the functional properties would only require a split of (0, 1) and (2) as this would provide full coverage as a complexity measure and would let the tests (0) and (1) be different from (2) according to the coverage metric, as they only test the stopping condition, still opening the search space to be controlled via a dynamic fitness function.

Two algorithms are presented that utilize a dynamically increasing fitness function to control the fitness landscape. Both work by producing building blocks on a less restricted landscape and continuously adding test cases to utilize the existing building blocks in later generations when the selection pressure is increased by adding more test cases. The first approach sequentially adds more and more tests to direct the population growth. The second approach groups tests in parallel populations, which are combined in larger populations containing building blocks that can be combined to satisfy the entire test suite.

Sequential Fitness Evaluation

Considering a sequential increase of the fitness of a function is done via the *complexity* measure of test cases. The core assertion being that test cases with a smaller complexity score are easier to achieve from an algorithmic viewpoint. Sequentially adding more and more test cases during a GI experiment allows the search space to slowly mature towards a more easily achievable goal, and then improving this goal towards the next step in complexity.

The reason the sequential fitness function uses the *complexity* measure and not the *overlap*, is that the complexity measure naturally ranks the tests in a suite. Splitting testing groups by overlap often coincides with complexity in smaller ASTs, but in larger ASTs less complex tests often have little overlap with each other. Grouping those together via complexity would defeat the purpose of the sequential fitness function. In contrast, grouping by overlap, but not complexity, would not achieve the goal of incrementally tightening the valid search space.

Figure 3.2 shows how a genetic algorithm behaves when utilizing a fitness function, iteratively increasing the complexity, and therefore the selection pressure, during the execution. The algorithm works by a given *complexity* measure, that a test suite is supposed to be ranked by. The *AST* that needs to be optimized is instrumented, and the *complexity* measure is evaluated for each test, which is then ranked and assigned into one of n groups. Instead of utilizing the entire test suite from the start, the algorithm starts with the "simplest" test cases according to the given metric and runs for several generations. After each step, the test group is merged with the next increasingly complex test group, and the algorithm is continued in a new iteration with the now more restrictive fitness function. This fitness function obviously must contain the *accuracy* measure (see Algorithm 1), but can also be geared towards one or more NFP at the same time.

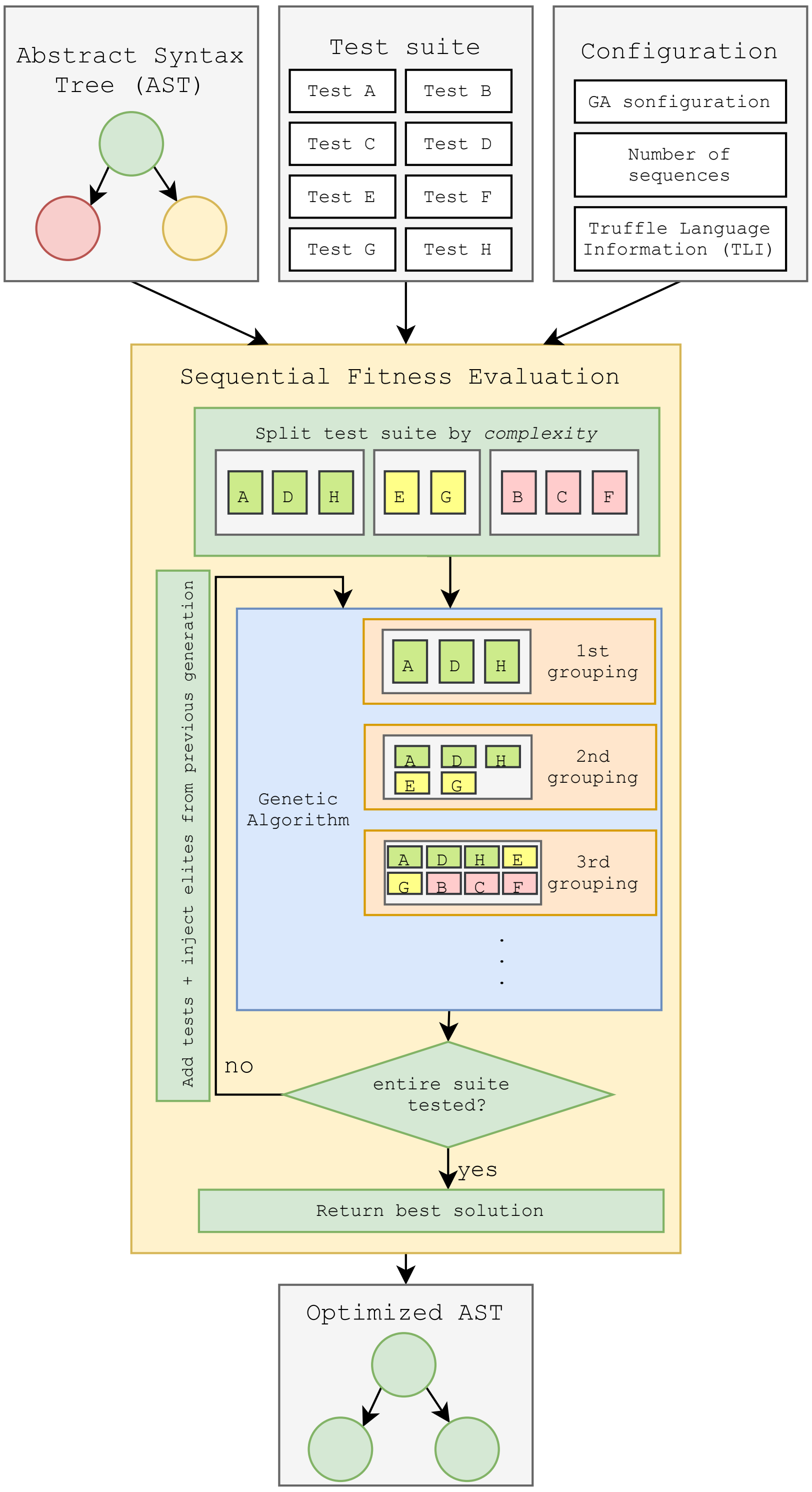


Figure 3.2: Genetic Algorithm with multiple repeats, iteratively increasing the tests used in the fitness function to verify the semantic validity of the individuals being optimized in the population. The tests are ranked and grouped via a given complexity measure.

The algorithm only carries over the top n individuals in a population to the next generation, and newly seeds the rest of the population. The reason for this is, to prevent the entire population from being trapped inside the local optimum for the previous fitness function. This inserts some individuals that provide good candidates for crossover and mutation with newly seeded individuals to break out of the artificially introduced local optimum.

As an example, assume that the sequential fitness function is not used in the context of GI but rather for GP. Starting from no code at all, the experiment shall generate a valid implementation of Fibonacci, according to the test set (0, 1, 2, 3, 4, 5) with their respective outputs (0, 1, 1, 2, 3, 5), according to the *complexity* measure *node coverage*. Considering the AST nodes in Figure 3.1 the assigned *complexity* values are (10, 13, 21, 21, 21, 21). When splitting these into several groups the tests would be (0), (1), (2, 3, 4, 5). This allows the GP to only attempt to return the input if it is 0, immediately creating the stopping condition. It also shows that the approach has limitations, as the added value of 1 does not represent another feature, and most likely the GP experiment will just satisfy returning the input. A valid set of individuals in the population is already provided when starting the third iteration with (2,3,4,5) which then increases the selection pressure with the actual implementation of the Fibonacci sequence.

An evaluation of the algorithm [73], has shown some limitations of the sequential approach, as the "sudden" increase in the selection pressure after several generations can lead to the algorithm becoming stuck in local optima. Two possible methods could improve the algorithm. The first is a specific mutator that attempts to create mutants of already good solutions in the previous iteration before the next iteration starts. This concept is comparable to the approach of grafting from GI. The other is extending the genetic algorithm from very distinctive iterations that run for several generations with just one fitness function towards an Age Layered Population Structure (ALPS) genetic algorithm. This type of algorithm allows the crossover and mutate operation over several past generations that were already produced, instead of just the last one. This would allow a smoother transition from a less restrictive fitness function to a more restrictive one as there is a transition period where the older population structures can still be taken from the previous optimum and crossed or mutated with the new optimum [77].

Parallel Fitness Evaluation

The parallel fitness evaluation, as implied by the name, requires splitting the population of a genetic algorithm in several closed off groups that are evaluated in parallel. Every single one of these populations is evaluated via a different fitness function. These fitness functions must contain the *accuracy* measure (see Algorithm 1) and evaluate different groups of tests. The grouping of these tests is done via the *overlap* (Equation 3.4) function. Similar to the concept of the sequential approach, this separates the search space into smaller, more manageable portions that have more leeway in what is a valid solution.

The approach is steered by *overlap*, as the core assumption is, that tests that cover similar code regions / branches also cover similar functional features. Unlike the sequential approach via *complexity*, this means that the test cases in the same group may be more varied in how complex they are. This also implies that a secondary ranking via complexity and a possible combination of the parallel and sequential approaches are feasible. However, this requires a larger amount of test cases that are different in nature in the region of the AST that the GI experiment is supposed to optimize, and thus a combination of both features is not attempted.

The approach to utilizing a fitness function that parallelizes test cases is shown in Figure 3.3. A given test suite is executed on the original AST and the overlap of nodes between all test cases is calculated. Overlap is only valid between two given tests. Thus, the grouping happens by creating groups, and randomly selecting the first test in each group. Via round-robin, each group is assigned the next test, by the test that has the most overlap with the first test in the group. The algorithm then creates a population for every group and conducts a regular genetic algorithm approach for that population and test-case pairing. After several generations have passed, the amount of groups is reduced, and the tests from removed groups are assigned via the same round-robin approach to all remaining groups until all tests are assigned a new group. This is conducted until only one group is left, covering all test cases.

After every repetition that reduces the groups the population is re-seeded, similar to the sequential approach, in order to reduce the likelihood of getting stuck in a local optimum. The seeding happens with elites from all groups that have provided test cases to the new larger group. A part of the population is filled with newly created individuals. This is similar to the sequential approach.

To better emphasize how the algorithm works, consider a GP experiment attempting to create the Fibonacci sequence with the test set of (0, 1, 2, 3, 4, 5) with their respective outputs (0, 1, 1, 2, 3, 5). Omitting a random assignment and round-robin, it is obvious to a human, that the tests (0, 1) and (2, 3, 4, 5) have the best overlap options, where 0 and 1 represent the stopping condition for the recursion, and 2 through 5 calculate Fibonacci via recursion. This essentially represents the two core building blocks of the original function, and lets the algorithm recreate these two blocks in parallel, and then merge them.

The example also shows one limitation identified in the original publication [73]. The parallel approach carried the issue, that very distinct building blocks often were not combined in a meaningful way. A specific crossover operator, attempting to merge the elites from the respective originating groups via different control structures, could improve this flaw. An additional option for improvement would be the utilization of Island GP. This type of algorithm already defines the use of several populations that evolve in parallel, simulating evolution via natural borders. In some cases, individuals traverse between these islands and would thus introduce relevant building blocks to them [78].

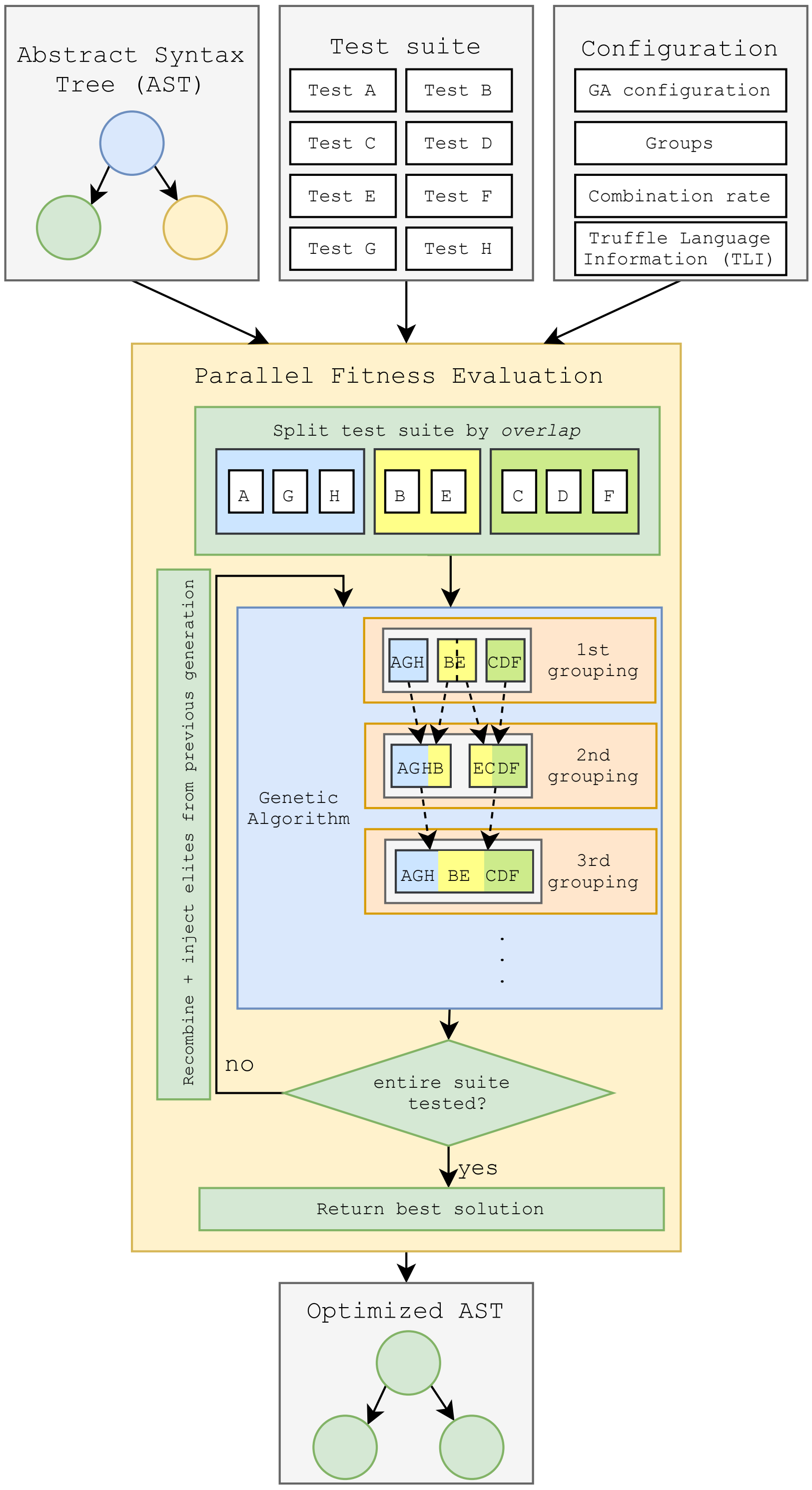


Figure 3.3: Genetic algorithm maintaining multiple populations with different fitness functions at the same time. Every repeat it reduces the amount of populations by merging them and combining the test cases driving the semantic validity in the fitness function.

Usefulness for Pattern Mining

Both of the presented options for dynamically applying fitness functions unfortunately present a major flaw that makes them unsuitable for mining patterns in the NFP domain. How the test cases are ranked depends solely on the existing implementation of the AST. This naturally encourages the population produced by the genetic algorithm to be similar to the original AST, and thus negatively influences patterns that can be mined, as the differences between the original and new versions will be lowered.

This issue does not occur in the functional domain, i.e., when fixing bugs. Quite to the contrary the ranking mechanism, especially via *overlap* encourages evolution in very specific areas, meaning that locations containing bugs are moved into the focus of some population groups, which increases the likelihood that localized fixes for them will be found, and patterns can be mined from these fixes.

3.4 Knowledge-guided Genetic Improvement

A large part of a population during a GI experiment will not compile or produce run-time exceptions [8, 9, 26] . Knowledge-guided Genetic Improvement (KGGI) specifically addresses this issue by first gathering knowledge about a language being used in GI, and then applying this knowledge in the form of a syntax graph that can be utilized to produce AST in a way that excludes incorrect individuals in the search space.

KGGI is based on two existing implementations of genetic programming. Grammar Guided Genetic Programming (GGGP) [27, 28] utilizes a provided grammar of a programming language, and was developed to improve the original crossover operator of GP [14]. GGGP only considers crossover points that are syntactically correct according to the provided grammar. Tree Genetic Programming (TGP) utilizes a tree structure, usually an AST, to represent individuals, and base their genetic algorithms on this representation form. TGP enables useful operators such as the homologous crossover, e.g., crossing individuals at similar positions, which can be easily applied to trees [25]. Both of these genetic programming variants have previously been combined via Tree-adjunct Grammar Guided Genetic Programming (T3GP) [79, 80], utilizing the tree representation for the operators, which can also use the syntactic information the grammar provides.

KGGI combines GGGP and TGP in a different way. It also uses ASTs as a representation form, but the core functionality is provided via a syntax graph that is synthesized from the grammar of a language. This syntax graph enables the valid selection and creation of ASTs, and enforces restrictions to the functional properties of software, i.e. preventing the creation of ASTs that knowingly will produce an exception at run time. It also allows restricting according to NFPs of code, e.g. restricting the size of the AST or preventing it from reaching an upper limit in predicted run-time performance.

Due to the application of this approach in a compiler or interpreter, KGGI does not use a context free grammar, but rather utilizes the context of a given AST under optimization. This enables restricting the search space represented in the syntax graph even further, for example with information available about variables, both global, and local variables in- and outside the subgraph. Information about the context, such as connected nodes and functions called in the individual AST can also be utilized.

Syntax Graph

At the core of KGGI is the *syntax graph* that can be utilized in all major genetic operators (create, crossover, mutate), as well as for selecting good positions to apply these operators (select). To achieve this functionality, the syntax graph provides several core features. The graph *restricts the relationships* between individual nodes in a given AST in order to enforce the given grammar, i.e., to be able to compile the AST. It also prevents ASTs known to produce a run-time exception. The syntax graph *restricts the non-functional search space* by asserting upper limits upon them via *knowledge encoded in nodes*. Finally, the syntax graph deals with *requirements*, i.e. several nodes require another node to not lead to a run-time exception. For example, a variable must be initialized before it can be accessed. This is dealt with in the form of requirements that the graph must satisfy whenever it is used to generate a new AST.

A syntax graph consists of nodes, just like an AST consists of nodes. However, in this case the nodes of the *syntax graph* represent operations that produce AST nodes. These operations are used in evolutionary operators in GI experiments. To ensure that these very different concepts can not be confused with each other, each concept is addressed with a specific term. Whenever referring to nodes of the syntax graph the word *strategy* will be used (Definition 3.4.1), as each syntax graph node follows a strategy to produce valid nodes. A *node* always refers to an object that is part of the AST (see Definition 3.4.2). An evolutionary operator is an *operator* (Definition 3.4.3) in the context of genetic algorithms, i.e. create, crossover, mutate, select.

Definition 3.4.1 *A **strategy** is a node in the syntax graph. Its purpose is to produce AST nodes, and to control the valid search space in the AST.*

Definition 3.4.2 *A **node** is a node in the AST. It is produced by a strategy in the syntax graph.*

Definition 3.4.3 *An **operator** is an evolutionary operator in a genetic algorithm. It utilizes the syntax graph to create ASTs.*

Figure 3.4 shows a reduced example syntax graph for the language MiniC (see Chapter 9). The syntax graph consists of a number of strategies, with one *root* strategy that has relationships to all other strategies. These strategies in turn are always *specific for one concept* in a given language, such as control structures (if, loop, ...) literals (int, double, ...) variable access (read, write) and others. Additionally, there is an *entry point* strategy, which serves to control the search space in the root of what is being created.

The syntax graph has three primary goals:

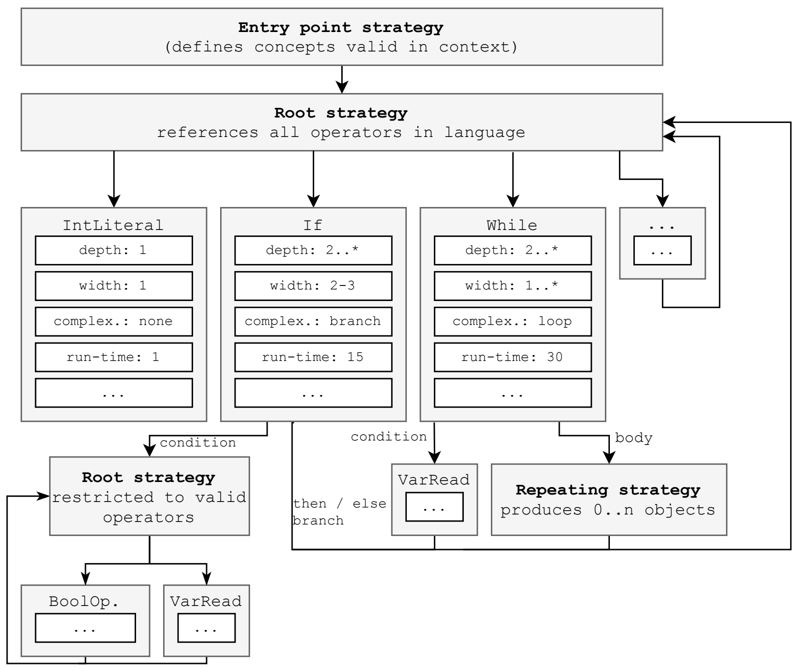


Figure 3.4: Syntax graph for a given language. Operators (gray) represent the grammar and contain knowledge about their non-functional properties (white), and edges to valid relationships according to the grammar.

- Create a compilable AST.
- Restrict the search space, i.e. decide what (sub) AST can be validly produced in a given context.
- When a branch in an AST is removed, produce a definition of what must be done to replace it to ensure the remaining AST stays valid.

An AST that is produced must always be considered in the *context* of its surroundings. These surroundings consist of the program the AST will be executed in. For example, the AST may call functions defined in the program, in addition to library functions the programming language offers. Considering genetic operators, especially mutation and crossover, only a sub-AST is mutated or crossed. Thus, the context must also include information about the surrounding AST the new sub-AST will be part of. As an example, a mutated sub-AST may be required to initialize a variable that is used later in the surrounding AST.

The *context* during the creation of a syntax graph consists of:

Approximated NFP of current values. These NFPs range from the depth and width of the AST to more complex information such as the approximated run-time performance. All of these values can only be seen as an approximation, as it is unknown how it behaves during run time. For example, the depth may be miscalculated as the interpreter or compiler may significantly modify the given nodes.

NFP limitations that restrict the search space in that aspect.

AST as it has been previously produced, mainly to access information for more specific strategies, such as those dealing with variable access.

Requirements that must be fulfilled, including the Degrees of Freedom (DOFs) remaining for later nodes in the currently synthesized AST to fulfill these requirements.

Supporting Information about the program the AST will be embedded in, such as the stack and heap variables, and available functions.

All strategies in the syntax graph follow some core rules to achieve these goals. Nodes are always generated with a given context, and the AST is generated in a depth first way, from left to right. This enables continuously updating the context during the generation, and thus allows later nodes to access the context to decide what is valid in that context. For example, the context contains already declared variables that may be accessed from that point on. The context is also provided by evolutionary operators, i.e., when only a part of a given AST is being modified, the context is pre-filled to contain the information of already existing variables, the depth of the tree that already exists, and so on.

The *entry point strategy* is responsible for preparing requirements imposed by the GI experiment, or alternatively on the evolutionary operator being used. Generally, this strategy will be restricted to all valid nodes that may occur under a node that is the root of a newly generated AST. For example, a function allows only the node type *function body*. In the mutation operator, the entry point depends on what is being mutated. For example, a single point mutator selects an *if* statement to be replaced. The *if* statement is the child of a *block (...)* node, meaning that at this position any statement node is allowed to be injected into the language, and thus the *entry point* would directly point to these nodes. In another example, if the *condition* node of the *if* statement, is selected for mutation, the entry point strategy impose the requirement that the generated AST produces a boolean value.

The *root strategy* is the reason why KGGI utilizes a syntax graph and not a tree to represent the language. It serves as selection mechanism for all strategies in a given language, whenever a strategy needs to create child-nodes. Representing a language as a tree would represent an exponential amount of possibilities, and even when restricting the space to a specific depth, it would be too large to be utilized in a meaningful way. Thus, most nodes, instead of pointing towards all allowed sub-nodes for a specific relation (such as if→condition), they instead point to the root strategy. The root strategy is given the context that is currently being generated or searched for, and it has to identify child strategies that are allowed to produce an AST node in that context, then select one to produce a new node.

The *root strategy* serves as a selection mechanism which node should be created. Whenever a strategy asks the root strategy for the creation of a new node, the root strategy queries all available strategies, providing the current context. Every strategy must decide if it can produce a valid AST given that context. The root strategy then selects one strategy that can produce a node given the context. This is also the reason, why in most cases strategies will not refer to specific child relationships, but just to the root strategy. To restrict the search space, instead of a specific link, a new *root strategy* (see bottom left of Figure 3.4) can be injected that contains only the valid subset of nodes.

Finally, to produce a valid node, *strategies* are specific for a given concept and depend on the grammar of the language. For example, via mining the given grammar, the required child relationships of a node can be decided automatically. However, in some cases there must be a specialized strategy. For example accessing stack or heap variables for concepts that deal with data flow have to be manually designed. Specialized strategies already help prevent many bugs that would otherwise occur in individuals generated during GI experiments. A specialized strategy for variable access can check which variables were initialized by the previously generated AST nodes, and thus only generate reads that will not fail.

There is one more strategy dealing with the occurrence of 0..n relationships between AST concepts. For example this happens in block statements ({...}), which allow multiple statements to be placed inside them. To avoid having to provide strategies that must consider a single occurrence or a repeated one, the *repeating strategy* is utilized. The repeating strategy occurs as a child of every strategy having such a relationship to child nodes. The reason being that the amount of repetitions can be controlled in a more fine-granular way, as there often is a difference between 0..n and 1..n to be valid for compilation or to prevent run-time exceptions.

Terminal strategies are a specialization to inject literal values. These can be, for example, variable names, integer or double values. Good starting points for terminal strategies are frequently occurring constants such as 0, 1 or null. Also, selecting constants existing in the original AST being optimized with GI or other parts of the source code via grafting can be good selection strategies [22, 23].

At the core of these strategies is the concept that given a *context*, a strategy must decide if it can produce a node that is valid in that context. This decision-making process is based on the properties of that strategy derived from the language's grammar and knowledge gathered about it. These properties are entirely non-functional and always restrict the search space. For example the depth of the generated AST may be restricted to a given value. An *if* statement knows that it has a minimalDepth of 2 - itself, and at least one node for the condition. It can check if that depth would be exceeded otherwise. Similarly, the maximum amount of child-nodes, or the complexity can be restricted. For example, the amount of loops can be restricted to a maximum, or the predicted run-time cost of nodes can be restricted as well. Restricting the allowed child nodes, can also prevent bugs. For example, a frequently recurring issue with generated conditions in loops or if statements is that the condition is generated as a boolean literal, always selecting one branch, or worse resulting in an endless loop. This type of bug can be fixed by constraining the allowed child nodes in the condition to not allow literal or constant values.

Restricting node creation to a valid range via the given context alone is not enough, however. A very important concept concerning bugs is a *fault of omission* [64]. This is a type of bug that is not introduced by code that exists, but rather a bug that exists because code was missing. This can include faults in defensive programming style, e.g., checking for null values, or the failure to check for negative numbers as seen in Figure 3.1. It can also include issues such as not initializing variables correctly or misusing an API which requires a specific order of calls [31].

The syntax graph also contains a fix for faults of omission with the concept of requirements engineering. Whenever a strategy requires another strategy, it can inject a requirement into the context being processed, which a node being generated later must fulfill. For example, a while loop may enforce that it's condition has a read to a variable to ensure that an endless loop becomes less likely. An important requirement that will be injected into the context is that the variable being read in the condition must be modified in the loop.

Only injecting requirements into the context in which an AST is being created, is not enough to prevent infeasible candidates from being created. The question remains at which point a given requirement must be fulfilled. Continuing the loop-example, modifying the variable value represented in the stopping condition may happen at any point during the loop body. Only defining the requirement would either enforce injecting it at the beginning or at the end, with a chance that multiple requirements then start interfering with each other. Thus, requirements have a corresponding *degree of freedom*. This is calculated by a look ahead in the generation, to see if any open space in the AST generation can satisfy one or more of the given requirements. The more AST nodes are generated in the process, the lower this degree becomes, and as a consequence, the more likely the requirement will be fulfilled, until it must be fulfilled as no degree of freedom remains.

Listing 3.1: Source code represented as AST in Figure 3.5

```
int sum(int[] arr, int len) {
    int i, sum = 0;
    for (i = 0; i < len; i++){
        sum += arr[i];
    }
    printf("calculated sum of
    %d values", i);
    return sum;
}
```

In the rest of this section, the core algorithms that drive the selection and creation process in the syntax graph will be explained. As these algorithms operate recursively on the syntax graph and consider multiple restrictions and requirements at the same time, they are fairly complex to explain. Thus, a running example shown in Figure 3.5 will be used.

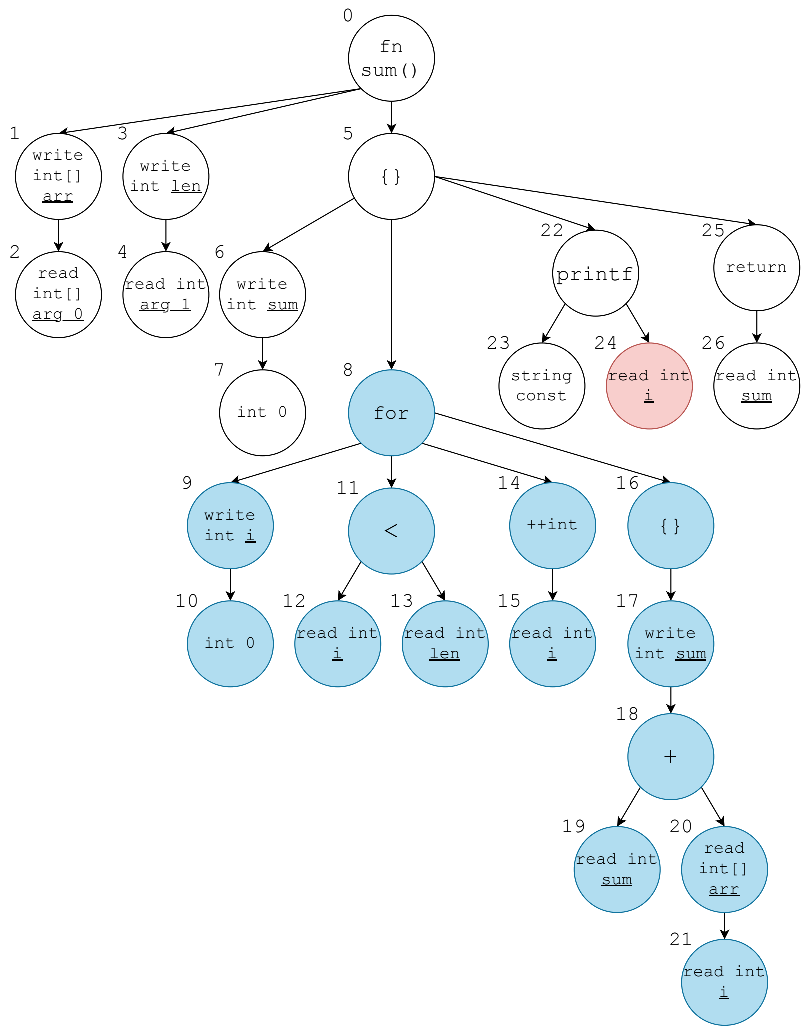


Figure 3.5: The shown AST is being modified by a single point mutator utilizing the syntax graph. The highlighted part of the AST is the subtree being replaced with a new mutant. The requirement that i must be written to from the node marked in red must be fulfilled in the new mutant to be valid. As sum is initialized in node 6, node 26 is not a requirement.

The code being mutated is a simple algorithm to calculate the sum of a given integer array. The example shows a case where an existing AST is being mutated via a single point mutator. The highlighted part of the AST was selected for replacement, and as a first step the context that a new mutant must fulfill to be valid has to be identified. For the example, we ignore NFP limitations and only concentrate on the AST and the requirements.

Deriving the Context From an AST

Before the syntax graph is applied to produce a new sub-AST for node 8 in Figure 3.5, the context must be initialized. The calculation of the context is shown in Algorithm 2. It iterates through the AST in a depth first way (Lines 4-17) and considers node 8 (Figure 3.5) as the *cutoffPoint*, e.g. the point that is cut out of the AST, possibly leaving unfulfilled requirements (Line 5). The entire AST is iterated through from left to right. Every node can add requirements, or fulfill them. For example, every *write* nod (1, 3, 6) adds the variable being written to, to a list of initialized variables. *Read* nodes (24, 26) can check if their requirement that a variable exists is satisfied. When this is not the case, as is shown with the node *read int i* (24), a requirement is added to the context, stating that variable "i" must be initialized (Line 15-16). Nodes can also fulfill requirements opened by other nodes (not shown in the example). For example, a *write* node may create a requirement that it will be read, to avoid dead code, and a later *read* node can fulfill this requirement.

Algorithm 2: Algorithm to identify context for AST creation.

```
   Data: ast
   Data: nonFunctionalProperties
   Data: cutoffPoint
   /* Initialization                                          */
 1 cutoff ← false;
 2 dfsIterator ← ast.dfsIterator();
 3 context ← {};
   /* Depth first search of requirements and NFP              */
 4 foreach node ∈ dfsIterator do
 5     if node = cutoffPoint then
 6         cutoff ← true;
 7         context.cutoff(nfp, strategy.nfp(node));
           /* Skip excluded part of tree                      */
 8         dfsIterator.skip(node);
 9         continue;
10     end
11     strategy ← findStrategy(node);
       /* Add non-functional properties and requirements      */
12     foreach nfp ∈ nonFunctionalProperties do
13         context.update(nfp, strategy.nfp(node), cutoff);
14     end
15     context.addRequirements(strategy.requirements(context, node));
16     context.fulfillRequirements(strategy.fulfill(context, node));
17 end
   Result: context
```

It doesn't matter if the reading happens from left to right or from right to left, as earlier nodes can impose requirements, and later nodes can have requirements that were only fulfilled in the removed part of the AST. Going from left to right however enables re-using the context. Everything of relevance is added to the given context, including, for example, the currently available stack variables. Lines 5-10 of Algorithm 2) denote where the cutoff point happens (e.g. the node including all child-nodes that will be removed from the AST). Line 7 marks the cutoff point in the context. This is important, as the later node creation algorithm must know what is available in the context for its use, e.g. which items are available on the stack and the heap, or which anonymous functions have been declared at this point.

Lines 7, and 13 (Algorithm 2) serve a similar context. They update the non-functional properties in the context relevant for the creation of the new subtree. Some of these properties are only relevant at the cutoff point, like the depth, which states that the maximal depth may not be exceeded descending from that point. Some others are relevant only for some remaining nodes, for example the maximum width (how many children a node has), and finally, some may require calculation over all nodes. An example of this would be the projected run-time performance, which has to be calculated for every node.

The *findStrategy* operation of line 12 (Algorithm 2) can be problematic in several cases. One case is, if there is no strategy that can create the node. In this case the algorithm has no choice but to ignore and skip it, as the strategy is responsible for identifying requirements the node has or fulfills. This may happen with artificial nodes introduced by the runtime environment, or with nodes not addressed by a developer. The other case stems from the design of the syntax graph. The syntax graph intentionally allows using multiple strategies for the exact same node. As an example, this is important for covering several valid versions of the same concept, e.g. multiple valid patterns identified via pattern mining. One while loop may require a write in the body, to update a read of a local variable. Another while loop may require a specific function call and a corresponding read to a global variable instead. This issue can be solved via pattern matching and is addressed in Chapter 5.

Looking at the running example the variables $arr[]$, (1) len (3) and sum (6) will be available in the context, and the NFPs will be set at a depth of 2 (depth 1 above the cutoff point node 8). The context at the mutation point will have one single requirement, that i (24) must be written to, as it is used in node 24, and it's corresponding writes (9, 14) are removed. Figure 3.6 shows the context that will be provided to the syntax graph.

Creating a New sub-AST With the Syntax Graph

The following example shows how the syntax graph works. In this case, a single point mutation operator uses the syntax graph to create a new node with the given context from Figure 3.6. The context will be updated alongside the example. For brevity, only changes made to the context will be shown, and the AST will only be shown from the cutoff point and below.

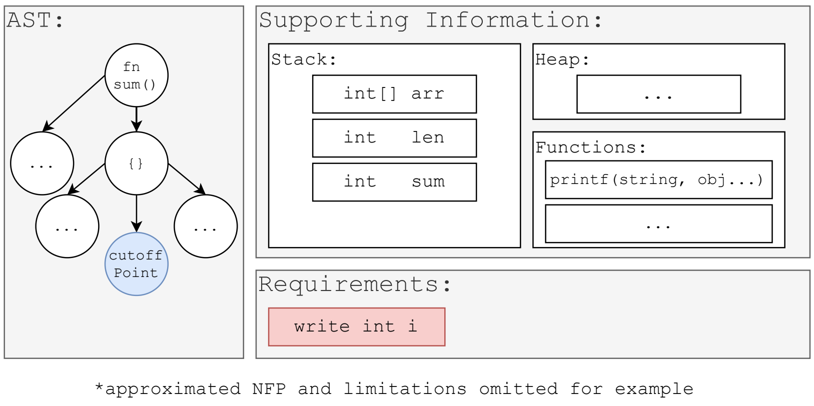


Figure 3.6: The initial context for the AST shown in Figure 3.5 generated from Algorithm 2.

The relationships between the strategies in the syntax graph are shown in Figure 3.9. It shows the core strategies employed in the syntax graph and their respective functionality with *canCreate*, which validates the NFP and requirements, as well as *create* which will return a new AST.

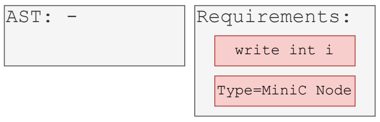


Figure 3.7: Context after entry point strategy.

The mutation operator calls the the *entry point strategy* ❶, which will check what type of node must be created. It checks the cutoff-point and identifies its parent, e.g. a block node ... (5). Then the entry point strategy will be asked to create a new node. Through the given context, that node will be of the type *MiniC node*, as a block node consists of a sequence of nodes, without a specific type. Thus, the *entry point strategy* requests the creation of such a node with the context (Figure 3.7) from the *root strategy*.

The *root strategy* ❷ then queries all of its contained strategies with the given context (RootStrategy.create() Figure 3.9). The contained strategies return nothing if they can't satisfy the NFP, or alternatively the given DOFs they provide in satisfying the given requirements. The DOF describe how often a requirement can be satisfied. This is used as a metric in the syntax graph, to enforce that all requirements are met, but not enforced immediately. For example, consider the requirement *write int i* which enforces that the variable *i* is written to, before it is used. To satisfy this requirement, the root strategy would have to create a *write int* node, essentially restricting the search space to just this one type of node. Instead, all *specialized strategies* are queried by the root strategy (canCreate ❸). A specialized strategy can either satisfy a requirement, or check if it can generate child nodes that can satisfy the requirement instead. In the case of a *write int* node, the DOF is 1. In the case of a *block* node, the DOF is 5. The block node, could not directly satisfy the requirement. But a block consists of a sequence of statements (in this example limited to 5 statements at most), and would delegate the query to the *repeating strategy* ❹. Since the block node could potentially contain 5 *write int* nodes, the DOF is 5.

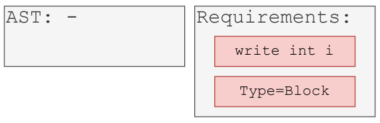


Figure 3.8: Context after the first node type was selected by the root strategy.

After all strategies have been queried, one strategy that can fulfill all requirements is selected. This selection is done just like how regular GI algorithms would select a node, for example randomly, via tournament selection, or biased towards a specific type of algorithm. The selected strategy is then tasked to conduct the actual creation. For our example, the {...} *block* node was chosen (Figure 3.8).

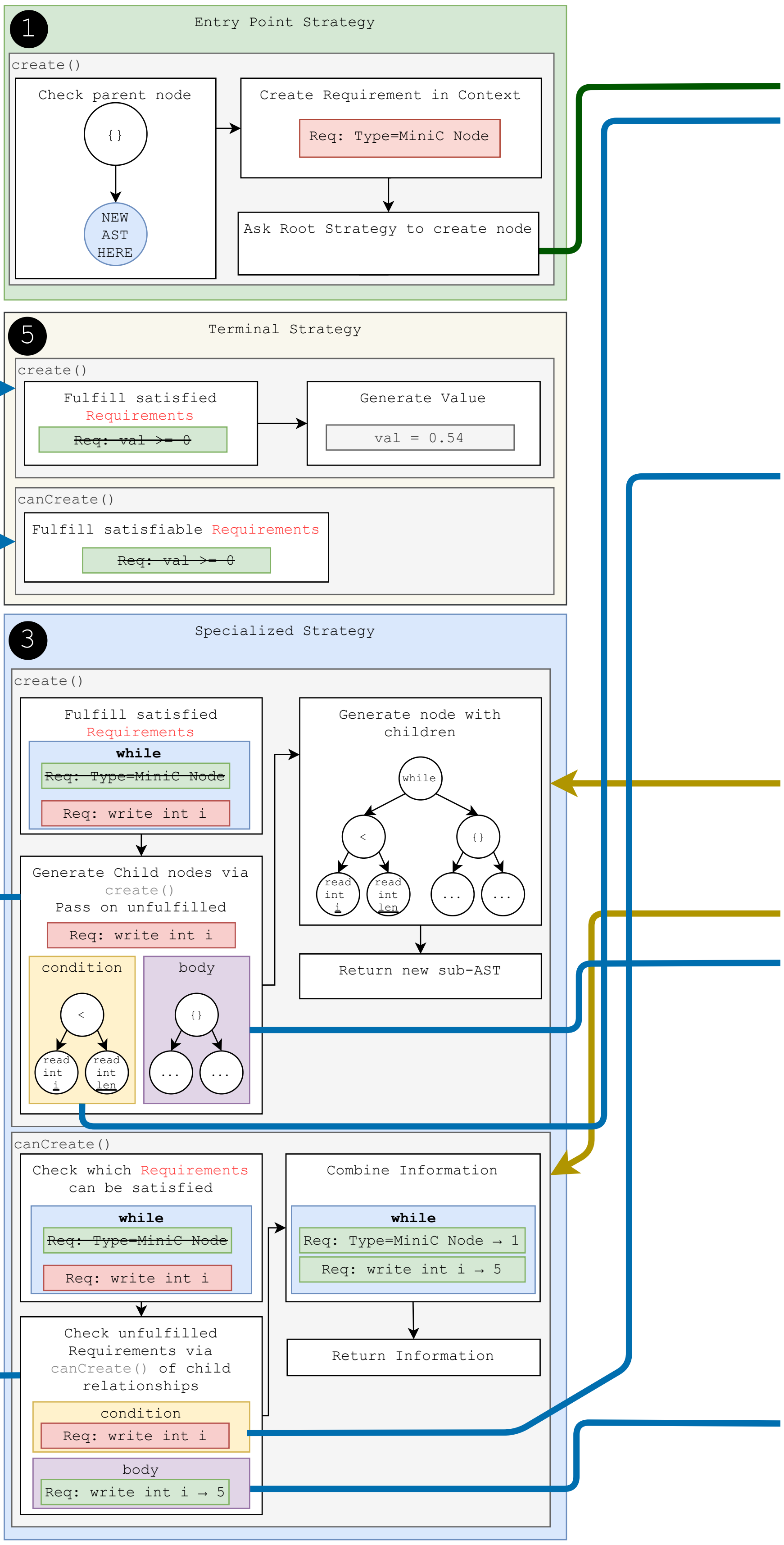
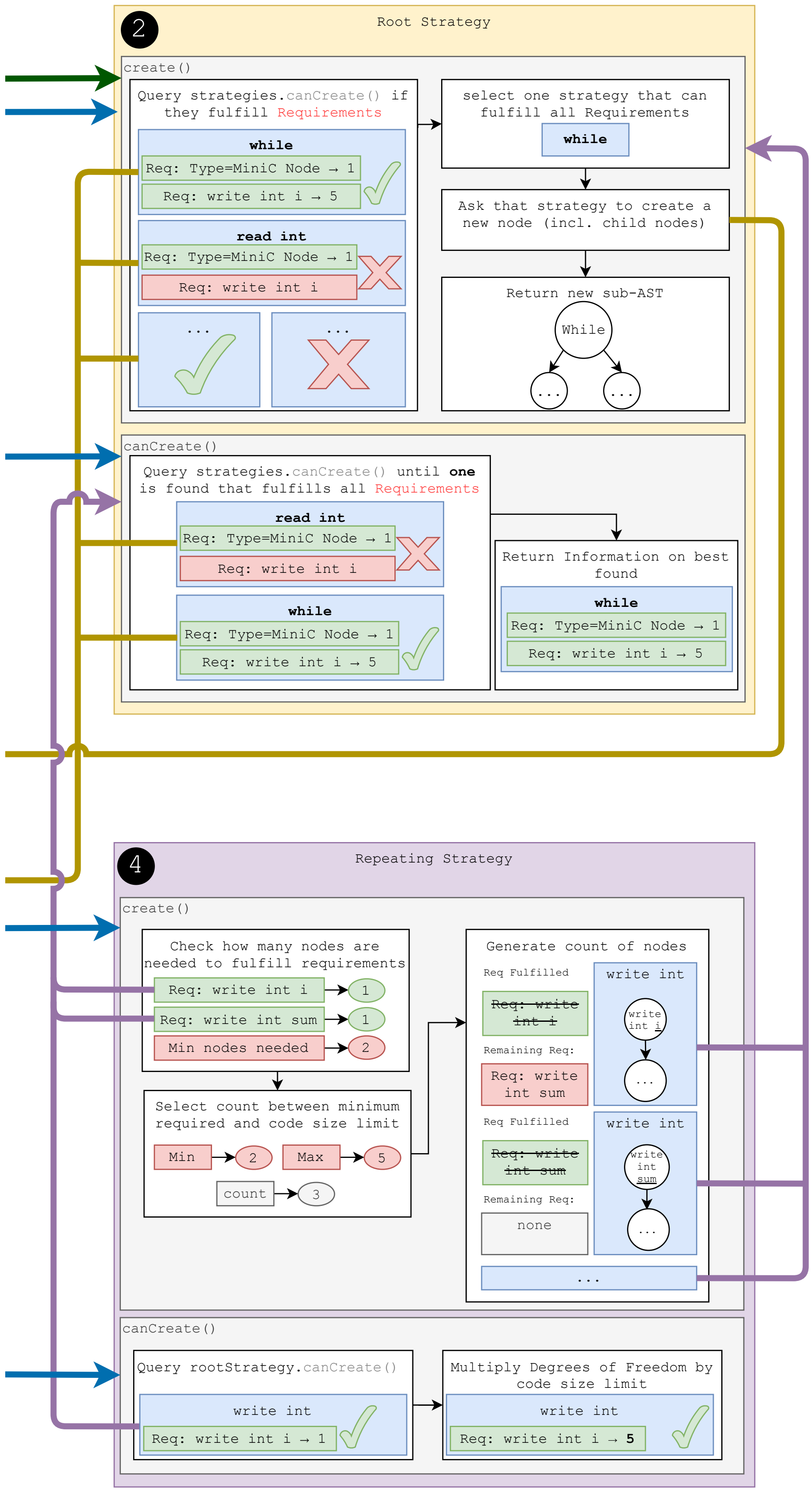


Figure 3.9: Algorithms for the major strategies in the syntax graph. The boxes in white describe the operations, and give examples of how this operation works.


2
Root Strategy
create()
Query strategies.canCreate() if they fulfill Requirements
while
Req: Type=MiniC Node → 1
Req: write int i → 5
read int
Req: Type=MiniC Node → 1
Req: write int i
...
...
select one strategy that can fulfill all Requirements
while
Ask that strategy to create a new node (incl. child nodes)
Return new sub-AST
While
...
...
canCreate()
Query strategies.canCreate() until one is found that fulfills all Requirements
read int
Req: Type=MiniC Node → 1
Req: write int i
while
Req: Type=MiniC Node → 1
Req: write int i → 5
Return Information on best found
while
Req: Type=MiniC Node → 1
Req: write int i → 5
4
Repeating Strategy
create()
Check how many nodes are needed to fulfill requirements
Req: write int i → 1
Req: write int sum → 1
Min nodes needed → 2
Select count between minimum required and code size limit
Min → 2
Max → 5
count → 3
Generate count of nodes
Req Fulfilled
~~Req: write int i~~
Remaining Req:
Req: write int sum
write int
write int i
...
Req Fulfilled
~~Req: write int sum~~
Remaining Req:
none
write int
write int sum
...
...
canCreate()
Query rootStrategy.canCreate()
write int
Req: write int i → 1
Multiply Degrees of Freedom by code size limit
write int
Req: write int i → 5

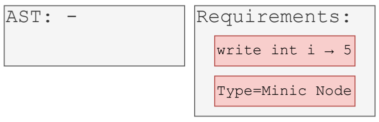


Figure 3.10: Context generated by the repeating strategy for the first node.

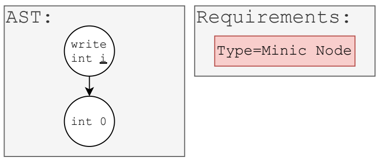


Figure 3.11: Context generated by the repeating strategy for the second node.

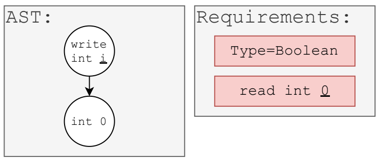


Figure 3.12: Context generated by the specialized while strategy for the condition node.

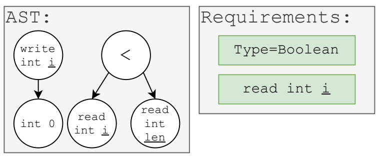


Figure 3.13: Context returned to the while strategy by strategy that created the < condition node.

The *specialized strategy* ❸ for *block* nodes first generates all needed child nodes. As a block node only consists of statements, it delegates the generation to the *repeating strategy* ❹. This strategy checks how many nodes are required to fulfill all requirements. In this example, at least one node is required. The repeating strategy randomly selects an amount of child nodes between this minimum, and the allowed code size limit (5). For this example, 2 is chosen. The repeating strategy then delegates the creation of the first node to the *root strategy* (Figure 3.10).

The *root Strategy* repeats the process of collecting strategies that can create in the given context, this time selecting the *write int* strategy. This would not have to happen, as the DOF are 5, thus allowing all nodes to be selected, and in this case just happens randomly. This *specialized strategy* ❸ for *write int* has one child relationship to the value that will be written to the variable. It also has a terminal, the variable name, that it will write to. For brevity, the generation of the child-node *int literal 0* is skipped (it repeats with root strategy.create() and a requirement on type = int node). Next the specialized strategy creates the terminal. In this case the strategy does not refer back to the root strategy, but rather to its *terminal strategy* ❺ of selecting a variable name available in the stack. As it selects *i*, the terminal strategy also fulfills the requirement *write int i*. This causes all later-produced nodes to be free of the requirement. The requirement is removed from the context and returned to the repeating strategy. The repeating strategy also updates the fulfilled requirements before it uses the root strategy to create the next individual (Figure 3.11).

The *root strategy* ❷ now creates the next individual. As no further requirements apply, the selection process is significantly faster, since not having to evaluate DOFs in requirements allows strategies to rely on the pre-existing knowledge encoded in them about the NFP alone. This time a *while specialized strategy* ❸ is selected. This particular strategy has been restricted via a structural requirement as shown in Figure 3.14. This structural requirement was injected via a pattern (explained in Chapter 5). The while strategy adds a new requirement to the context, requiring a boolean operator for the condition. Due to the pattern, the while strategy also adds an *read int variable 0* requirement (Figure 3.12).

The creation of the condition node is again delegated to the root-strategy. The root strategy randomly selects the *less than (<) specialized strategy* ❸ which fulfills the boolean requirement. This strategy in turn has two child nodes, left and right, either one of which must satisfy the remaining *read int variable 0* condition. For the example, assume that the left node generated is *read int i*, which also fulfills the *read int variable 0* requirement. The *less than specialized strategy* injects i in the fulfilled requirement, to replace the 0 placeholder (placeholders are numbered in case a pattern must be satisfied with multiple variables). The right node generated is *read int len*. This expresses the condition *i <len*, which is returned to the *while node strategy* (Figure 3.13).

The specialized *while* strategy ❸ next has to create the body. The strategy injects the second part of the structural requirement (Figure 3.14). As the variable *i* has been returned by the child context, this variable is now used in the new requirement *write int i*. The process continues until the entire body is created, which also contains a *write int i* node, satisfying the remainder of the structural requirement. The while specialized strategy

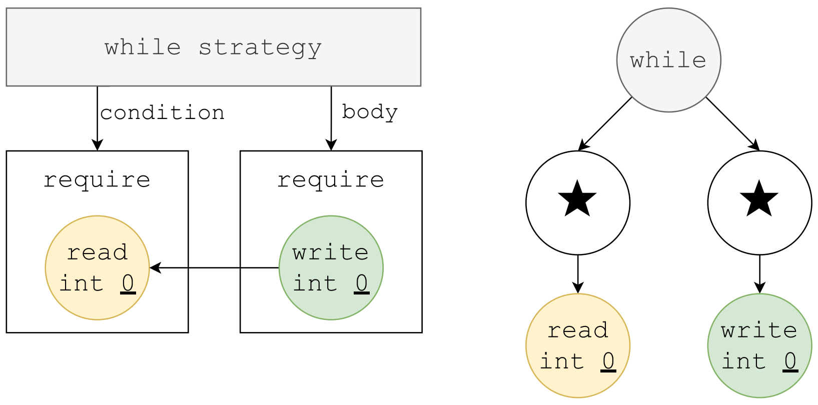


Figure 3.14: While statement strategy with a structural requirement that a local read is injected in the condition, and a corresponding requirement to write updating that variable in the body of the statement. The corresponding pattern is shown on the right side.

finally creates the *while* node, and links the already created child nodes *condition* and *body*. Then the process ascends back to the {...} *block* strategy which is finally created with the write to *i*, and the while node as children, finishing the process with a completed sub-AST.

The example also shows that the syntax graph is only as good as the rules that are encoded in it. The {...} *block* node contained inside the other block node is unnecessary, and could be avoided by encoding additional rules. The results of the process, without the redundant *block* node, are shown in Figure 3.15.

Listing 3.2: Source code represented as AST in Figure 3.15

```
int sum(int[] arr, int len) {
    int sum = 0;
    {
        int i = 0;
        while (i < len){
            sum += arr[i];
            i++;
        }
    }
    printf("calculated sum of
    %d values", i);
    return sum;
}
```

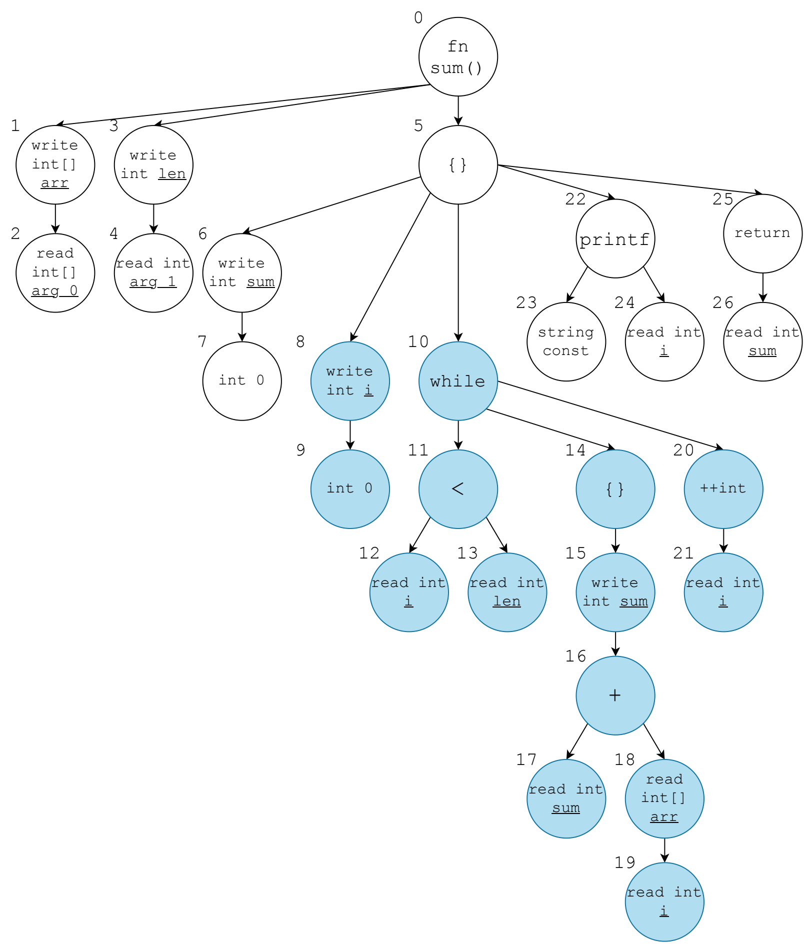


Figure 3.15: The shown AST after it has been modified using the syntax graph of KGGI. The highlighted part of the graph is the new subtree, fulfilling all requirements of the original example.

Collecting Information for the Syntax Graph

To enable the advantages that the syntax graph provides, it is necessary to gather the necessary information for a given programming language. This foundation can be utilized from the grammar of that language, and is highly dependent on the compiler or interpreter being used. In general, the Extended Backus-Naur Form (EBNF) can be utilized to build up the structure of the syntax graph. The primary difference is that the EBNF produces a parse tree whereas the syntax graph produces an AST (a more abstract representation of the program). In a parse tree, inner nodes are nonterminal symbols, while operators such as add (+) or multiply (*) are leaves. In an AST, however, inner nodes are operators and the leaves are operands (variables or literals). An AST does not contain nonterminal symbols of the grammar.

It is important to identify and extract down to the smallest operator or operand a programming language can have, and create a strategy for each one of them. In the case of the Truffle interpreter (see Section 2.4) this is fairly simple, as it is an AST interpreter, that already implements every possible operator as a specific node. For the operators of a language, how they are connected to each other is relevant, i.e., which operator can have which type of child nodes. This builds the basis for the syntax graph.

Additionally, the syntax graph has to be attributed with knowledge about NFP. These properties serve purely to restrict the search space to manageable levels. Some of these are fairly simple, such as depth (distance between root and leaf nodes) and width (distance from leftmost leaf to rightmost leaf). Some others must be identified and measured from code, or must be modelled. Advancing the state-of-the-art in this area is outside the scope of this work. Instead, specific to the Truffle interpreter, a code measurement approach is utilized (further discussed in Chapter 10). In short, this is done via Java reflection and by identifying the relationships between node implementations of a given Truffle language. NFP are then measured by a brute-force learning approach. This works via a method-stub containing a callable function, and a block statement where nodes can be injected. A node to be measured for NFP, such as run-time performance, is injected into that method stub 10,000 times to measure in-process iterations, and executed 1,000,000 times of which the first 100,000 times are discarded to take the warm-up phase into account. From the remaining executions, the NFP are calculated (e.g., average run-time over the 900,000 executions) to ensure that the side effects in the measurement are minimal [81]. As the operators in the language are evaluated independently of each other, the measurements cannot be seen as an absolutely trustworthy behavior of the code, as during compilation code will be optimized, which will for example change the run-time performance. They do however provide a useful upper limit to the search space.

An alternative to the selected approach would be modelling. Kommenda [82] uses a recursive complexity metric for mathematical functions depending on the type of expression used. Constants are assigned to leaves (e.g., variables and literals), and aggregation functions to inner nodes (e.g., operators such as + or square root). Using a modelling approach to approximate the real behavior of code is also done directly in compilers. Leopoldseder et al. [58] uses a cost model for the selection of code duplication optimizations in the Graal compiler. Yet another alternative may be predicting the NFP via unsupervised learning instead of a model [83]. However, this option may not be as easy to utilize in the syntax graph, as a predictor would have to be able to be applied on individual operators during the recursive creation of AST nodes.

Finally, the requirements needed for the syntax graph to prevent infeasible individuals, are also necessary. These often cannot be collected automatically, but must be hand-coded. For example, access to the stack and the heap are different in every execution environment and will be represented differently in interpreters and compilers. This requires a manual implementation. The only requirements that can be collected through semi-automated means, are structural requirements, such as shown in Figure 3.14. These structural requirements can be derived from patterns. An approach to mine and utilize these patterns is discussed in Chapter 5.

Operators in KGGI

The presented syntax graph provides the core functionality for KGGI, and can be applied in every genetic operator, though it is not necessarily required. The syntax graph can be applied in a single operator as well, mixing KGGI with other existing approaches. The following explains how the syntax graph can be utilized in every major genetic operator.

In the *creation* operator, i.e., the genetic operator that is responsible for creating the first individual population, the syntax graph alone can be the operator. Given the limitations of the selected experiment, a new AST can be created without any other information. Often *creation* operators in GI use mutation of the original AST, combined with the grafting from other parts of the source code, to create the initial population [22, 23]. This is a form of *mutation* operation, which can utilize the syntax graph via Algorithm 2, on pruned mutants of the original code, or alternatively grafts from other parts in the source code.

Finally, the *crossover* operation, which can also have a grafting adaption where an individual is crossed with ASTs from other parts of the code, is the most complex to handle. The syntax graph can be utilized for this via Algorithm 2 to identify requirements from one crossover point, and also requirements from another crossover point. Combining this with an optional mutation to create fixes for non-matched requirements in both AST candidates is one way to conduct crossover. The other depends on the *selection* operation. The syntax graph can be utilized to identify requirements for different candidate positions during a crossover. The selector, in this case, can use the generated context of Algorithm 2 to match crossover points that either produce no issues, or solve each other's requirements.

Advantages and Drawbacks of KGGI

The primary drawback of KGGI is its complexity, relative to other approaches. Not only does the syntax graph require a rather complex set of algorithms and needs specific functionality for measuring NFP, it also requires handling non-structural requirements manually. The recursive nature of the syntax graph strategies also has a rather large run-time overhead, as the mechanism for selecting which node is created requires evaluation of multiple nodes. This can be somewhat mitigated by a random walk that simply selects the first evaluated strategy that fulfills all requirements. This makes the genetic operators using the syntax graph less lightweight than more random approaches.

The primary advantage of KGGI is the reduction of individuals that are infeasible due to run-time exceptions and complete removal of individuals that can't be compiled, as the grammar of the language is adhered to at all times. Conversely, this negatively impacts the run time of a GI experiment from start to finish, as evaluating feasible individuals is more costly than evaluating infeasible individuals that produce exceptions. Section 6.4 shows that KGGI can double the amount of successful ASTs in an experiment and can generate solutions that significantly outperform manually written algorithms concerning their NFP run-time performance.

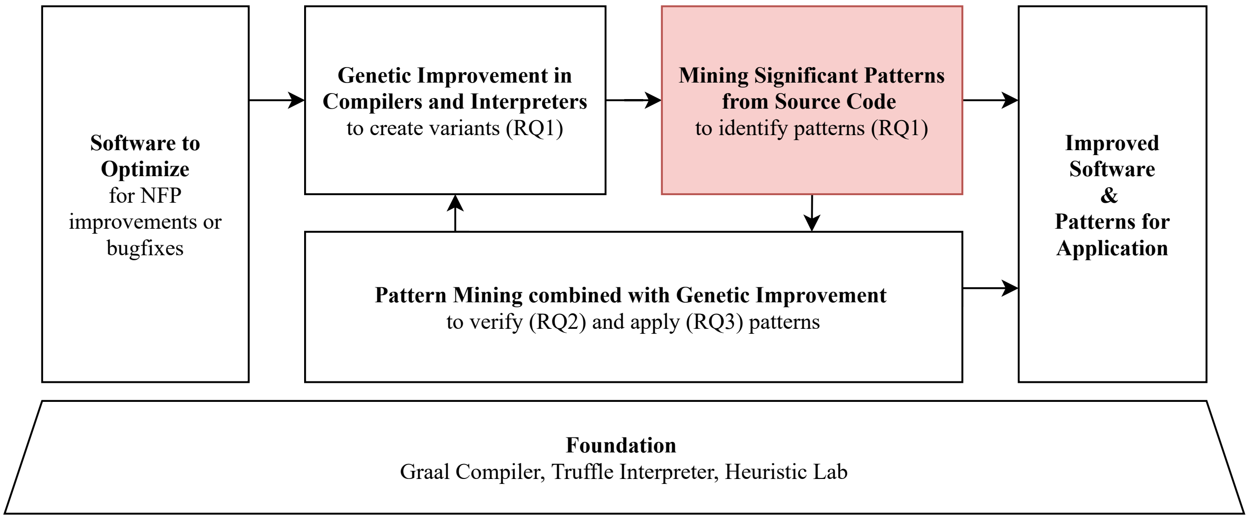
Mining Significant Patterns from Source Code 4

Patterns in source code can be discovered via mining the frequency of a section of code. This can be done *static*ally by mining the source code and by gaining information such as the data flow graph using static analysis [31, 34, 40–43]. It can also be done *dynamic*ally via analysis of how the code behaves by extracting information from call traces or execution traces [2, 37, 38, 45]. Of course these approaches can be mixed into *hybrid* approaches [44, 46].

This work follows a *hybrid* approach of pattern mining, using an Abstract Syntax Tree (AST) representation of the source code. While the source code is utilized in a static way, it is enriched with information gathered from the execution of that source. This information consists of the functional properties of that source code, i.e. the results of executed tests and exception traces if a run-time exception occurred, as well as the Non-Functional Properties (NFPs), such as the run-time performance and energy-consumption.

Pattern mining can be done via comparing multiple different source code fragments from different programs or frameworks. It can also be done on one single structure, such as large graphs, to find frequently recurring substructures. In the context of our work, the *search space* always consists of several different ASTs. This can be multiple algorithms being mined for recurring bugs (see Section 6.3), or multiple versions of the same algorithm represented as different AST, being mined for NFP (see Section 6.5).

Mining of patterns via the frequency of sections of code has a major drawback. The larger the code sections, and the more relationships between them, the more the search space will grow. For example, a single tree has 2^n components, where n is the amount of relationships in that tree. A component of a tree is any subset of nodes with its corresponding relations in order. This is similar to the permutations, in which all



Software to Optimize for NFP improvements or bugfixes
Genetic Improvement in Compilers and Interpreters to create variants (RQ1)
Mining Significant Patterns from Source Code to identify patterns (RQ1)
Improved Software & Patterns for Application
Pattern Mining combined with Genetic Improvement to verify (RQ2) and apply (RQ3) patterns
Foundation
Graal Compiler, Truffle Interpreter, Heuristic Lab

variants independent of order are considered. Evaluating every single component would require exponential runtime. Thus, *significant pattern mining* follows a bottom-up approach via apriori or pattern-growth algorithms, and considers a *minimum support threshold*, i.e., a threshold that requires a pattern to occur with a minimum frequency to be explored any further. Naturally patterns occur less frequently the larger they are, no matter if they occur over multiple trees or multiple times in a single one, thus efficiently pruning the search space. *Discriminative pattern mining* extends this concept by mining between two groups (e.g. efficient ASTs compared to inefficient ASTs) which allows one to mine patterns that occur more frequently in one group. The pruning can be done with a variety of metrics, most prevalent the *Information Gain* metric [36, 38]. Our work also follows significant and discriminative pattern mining, but introduces a novel encoding and algorithms with adaptions geared towards the intricacies of source code under consideration of its NFPs.

The following sections explain the foundation of the mining approach and extensions to the state-of-the-art such as the use of taxonomies, wildcards and a novel encoding used in the mining. Simplified examples are provided how these extensions can be used in a meaningful way. All of these parts are tied together in a mining algorithm utilizing pattern growth discussed in Section 4.6.

4.1 A Representation Form for Source Code

An AST representation was chosen for the mining approach, largely because of the advantages identified from related work, and the natural way source code can be represented as a tree.

Related work in this area focuses on three types of representation. Sequences [34, 42, 43, 46], mostly represent the source code itself, or log traces of function calls. With regard to the meaningfullness of patterns, sequences are generally outperformed by tree and graph representations, but have the advantage of a large background in text mining and highly efficient algorithms due to the simple representation. Graphs [31, 37, 38, 40, 44, 45, 47, 48, 50, 51] are used most often for dynamic information such as call graphs or data flow graphs, but also to record class relationships or method blocks. Their advantage lies in the ability to represent the strucure of the source code as well as the ability to deal with cyclic relationships. In sequential mining, to derive patterns from cycles, cyclic relationships are resolved in sequences by transforming recurrences to 1..n patterns if they are observed in traces. Similarly, trees can contain structural repetitions or can use annotated relationships to deal with repetitions, but can't represent cyclic relationships. The major disadvantage of graphs is the inefficiency of their mining algorithms, as graph isomorphism has to be dealt with due to the cyclic relationships. Finally, tree representations [2, 41, 49] are used to represent call trees or the source code as AST or a variant thereof. Due to the acyclic nature of trees, highly efficient algorithms have been developed for them [30]. Their disadvantage is, that due to their acyclic nature the representation is not as versatile as a graph representation, and algorithms developed for it can only work with graphs that are preprocessed by removing cycles.

Trees have the advantage of naturally representing source code and can be produced by a parser from the grammar of a programming language. Trees also can be transformed back into source code. ASTs are abstract versions of syntax trees that do not represent the parsing hierarchy, but rather the structure of the code in the form of operators and operands [7]. They are used in compilers as a high-level representation in the early compilation cycle [4], and interpreters use them as a primary representation [60, 61]. Utilizing ASTs as representation allows the recording of observed information directly from the compiler or interpreter. Additionally, patterns mined from this information can be directly applied in said compiler or interpreter for the optimization of NFPs.

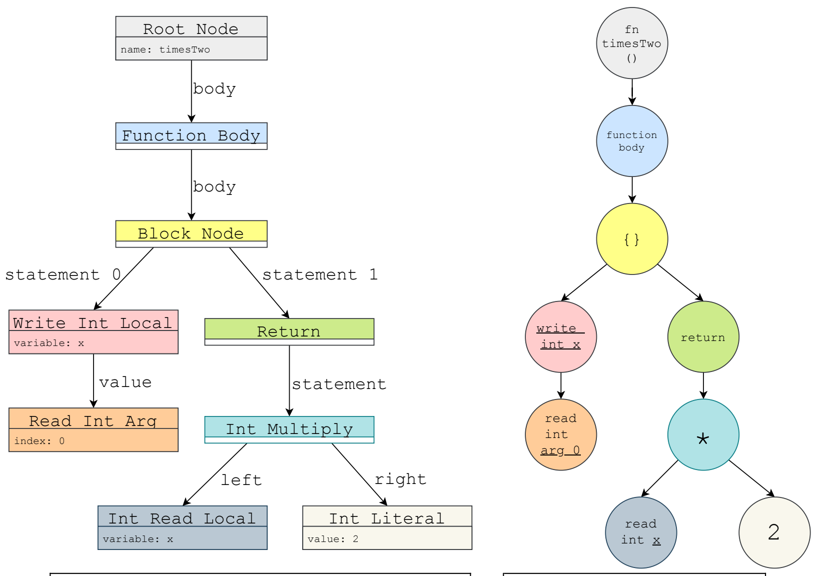
This representation can be used for a wide range of granularity. For example, when parsing source code via a grammar the code may be stopped at control structures (if, loop, ...), at structures that create branches (control structures, and, or, ...), or at the statement level. An interpreter may provide a much more detailed AST down to the expression or terminal symbol level. It may also provide this information with additional values, such call targets for function dispatches or the data type that a specific operation (+, -, assignment) will be executed with. While this granularity is a necessity to really understand the impact of source code on NFPs it may be a hindrance as well. Due to the detailed granularity, patterns also become more diverse, reducing the amount of patterns that can be found and also reducing the probability that patterns can be generalized. This issue can be dealt with by considering a taxonomy when mining a programming language. A taxonomy in this case represents a view that can contextualize mined patterns, such as a taxonomy dedicated for different data types operators and operands are written in.

Listing 4.1: Source code represented as AST in Figure 4.1

```
int timesTwo(int x) {
    return x * 2;
}
```

Definition 4.1.1 summarizes our representation for mining, which is extracted from the AST used by the interpreter or compiler. However, an AST can also be manually created from source code. As the order of operations in source code is significant, the names of relationships and their order in larger cardinalities is utilized in the mining process. ASTs in the context of a compiler or interpreter are always ordered, however, related work often conducts unordered mining on AST representations. In addition, values, such as the names of variables or literal values, are used for mining as well. ASTs are also enriched with NFPs and test results recorded during execution. Figure 4.1 shows an example of the data considered in mining on the left, and the simplified representation which will be utilized in showcases of the remainder of this chapter on the right.

Definition 4.1.1 *The* ***representation*** *source code is mined in, is an* order dependent AST. *The AST represents one method, including literal values and relationship names. The AST is enriched with* NFPs and test results *used for discriminative pattern mining.*



Test Results		
inputs	output	exceptions
{0}	0	-
{1}	2	-
{2}	4	-
{3.1}	-	Read Arg. Cast Exception.

Observations	
observation	value
code size	9
complexity	0
accuracy	0
run-time	9 ns

Figure 4.1: Representation of an AST. On the left the complete AST is shown, including literal values, and the names of relationships. If a relationship is a one-to-many relationship the order of the child nodes is used as well. The right shows the simplified version. Below the trees, the left table shows the test results observed during execution, while the right table shows the properties observed from static analysis (code size, complexity) and execution (run-time, accuracy e.g. if any tests failed during execution).

4.2 Utilizing Taxonomies in Pattern Mining

Programming languages and constructs follow a natural hierarchy that pattern mining can take advantage of. For example, a class can extend another class or implement an interface. This class consists of fields and methods. A method has parameters and a method body consisting of statements. As another example, a statement can be either a Return, If or Loop statement, and a Return statement consists multiple operators and operands. These operators may be of different data types, or have specific data access patterns. These hierarchies, or hierarchical taxonomies, of the programming language are not really represented in an AST. The AST represents the structure of the source code but not higher level concepts. Different layers in the taxonomy are important for mining, in part to maintain control over the granularity and thus the general applicability of a mined pattern, and also in part because depending on the property (run-time performance, bugs, ...) patterns are being mined for, some patterns may be combined into stronger, more generalized patterns and thus can be found with a higher *minimum support threshold*.

Thus, in our pattern mining approach a *hierarchical taxonomy structure* is utilized as defined in Definition 4.2.1. An example for such a structure can be seen in Figure 4.2. This taxonomy enables the generalization of a pattern along the hierarchy. An example of resulting patterns can be seen in Figure 4.3. These contain less frequent specialized patterns and more frequently occurring generalized ones. This results in an increased amount of possible patterns in the search space, adding a layer of complexity to the pattern mining. The amount of possible permutations increases from 2^n to $2^{(n*m)}$ where n is the amount of relationships in the AST and m is the amount of types a node can take. As seen in Figure

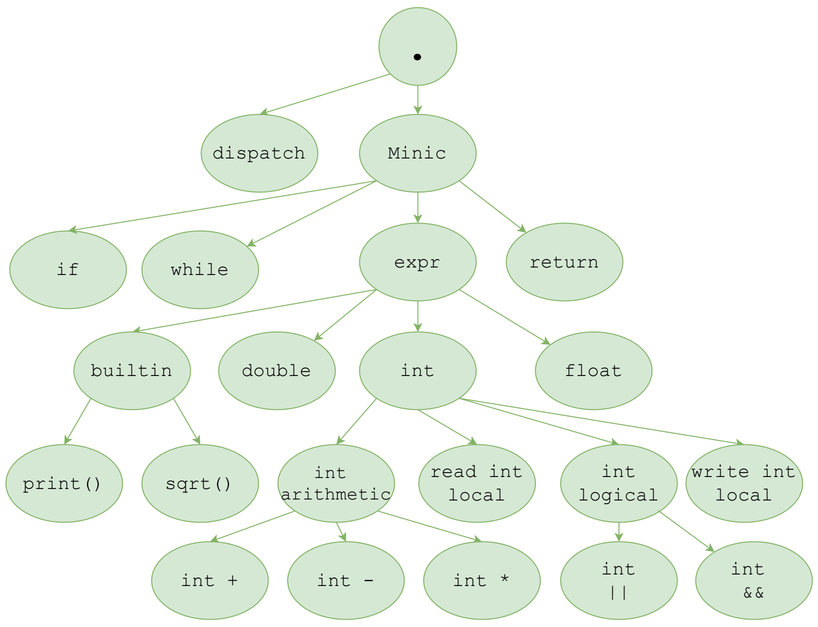


Figure 4.2: Taxonomy for the language MiniC, showing a small subset of the full taxonomy. The taxonomy is specialized towards the data types utilized in given operations.

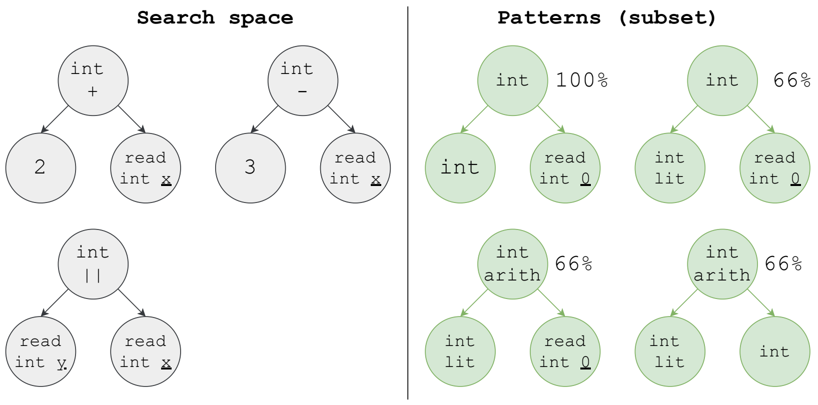


Figure 4.3: Patterns can be mined in a more generalized way by using taxonomies. The search space in the top half shows three ASTs that do not overlap in their root nodes. A subset of resulting patterns is shown in the lower half. Via generalizing *(int+)* and *(int−)* to *(int arith)* they can be combined into one stronger pattern. Similarly, raising the nodes further to *(int)* allows combining all trees into one pattern.

4.2 the amount of generalizations or specializations *m* can vary between specific nodes in a language. For example, two specializations of *builtin* compared to nine specializations over two levels for *int*.

Definition 4.2.1 *A **taxnomoy** is used in mining. This taxonomy is an* acyclic hierarchy*, that allows the generalization of individual AST nodes.*

The use of taxonomies also comes with a caveat. As can be seen in Figure 4.2, the trees with the root of *(int+)* and *(int−)* can be generalized to *(int arith)*. However, as the sub-nodes can also take multiple hierarchies the very same pattern is now represented with multiple abstraction layers in each node, with all of these patterns having the same strength of occurring in two out of three ASTs. While generalization can help to produce stronger patterns, it can also lead to an information overload in patterns, making the mining unhelpful for developers. Definition 4.2.2 says that the pattern that should be considered is the one which produces the strongest pattern, with the least amount of generalizations over all nodes. For example, if a pattern occurs exactly as often if one of its nodes is *(int−)* and the pattern would not occur more often if that node is generalized to *(int arith)*, only the pattern with *(int−)* is relevant. Of course, this increases the amount of patterns found overall,

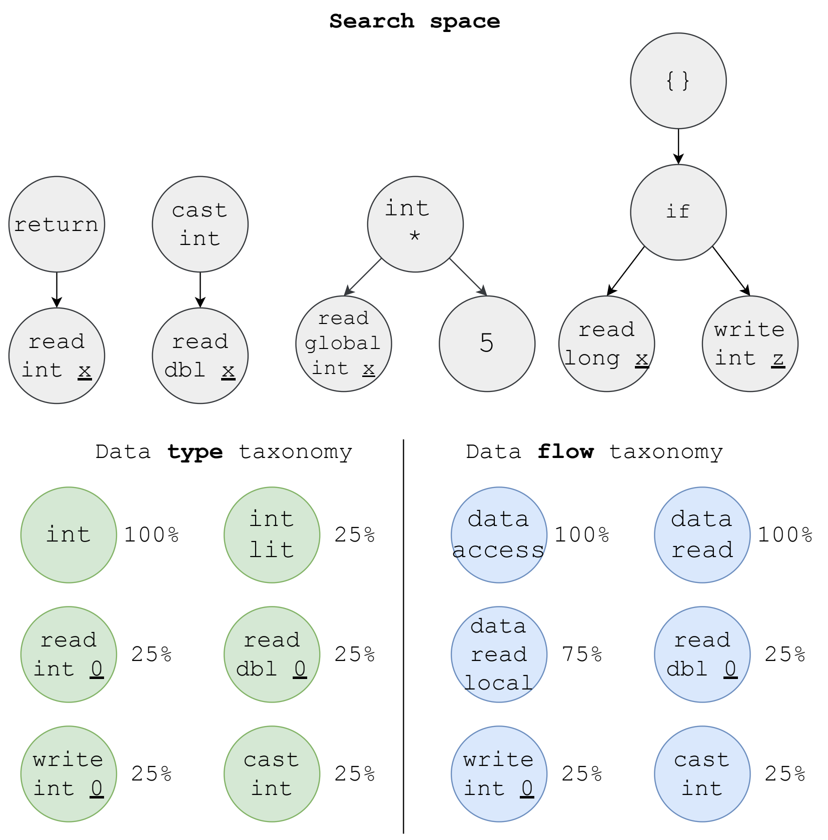


Figure 4.4: Within a given search space of four ASTs failing due to uninitialized variable access (top) two taxonomies are used for mining. The bottom left, mined with a data-type-specific taxonomy, shows the data type as the most relevant pattern. The bottom right, mined with a data-flow-specific taxonomy, shows the reading access as the most relevant pattern. The percentages show in how many of the ASTs in the search space the pattern occurs.

as with multiple combinations of ASTs in the search space, multiple generalizations in different nodes of the same strength can occur. For example, generalizing only the left side of an *(int arith)* allows the same amount of combined ASTs as generalizing only the right side.

Definition 4.2.2 *A given pattern is only **relevant** if no other pattern exists that has the same amount of occurrences and has fewer generalized, or less generalized nodes than the given pattern.*

It is also important to consider different taxonomies in the same language, representing different viewpoints that patterns can be mined for. These viewpoints are dependent on what goal the mining is supposed to achieve. For example when attempting to mine the NFP of run-time performance a taxonomy could employ special consideration for looping and branching structures. When considering the NFP of memory efficiency nodes accessing the stack or the heap may be considered, or alternatively the size of what is being allocated may be of interest.

An example is shown in Figure 4.4. This figure shows four different ASTs at the top written in the MiniC language (see Chapter 9). All of these ASTs fail for the same reason during execution, as the variable being read was never set, which according to the C standard is undefined behavior [84] and was defined in MiniC to lead to an exception. Consider the already introduced *default taxonomy* (see Figure 4.2) and a second specialized *data flow taxonomy*, which can be seen in Figure 4.5. Utilizing these taxonomies the patterns shown in the bottom of Figure 4.4 with size 1 will be mined (additional patterns omitted for size restriction). The left shows patterns with the *default taxonomy* whereas the right shows

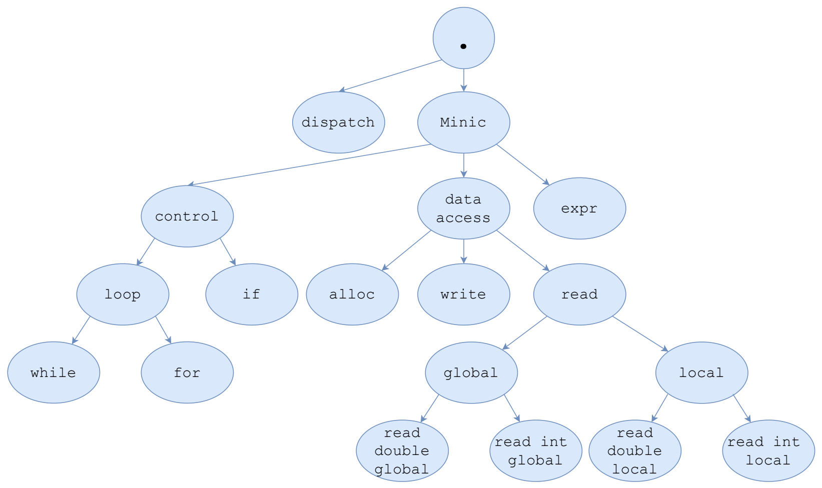


Figure 4.5: Taxonomy for the language MiniC, specialized towards mining patterns considering the data flow of ASTs.

patterns mined with the *data flow taxonomy*. The *data flow taxonomy* is specifically geared towards finding exceptions that happen due to access violations. They first generalize the data-type-specific nodes to nodes for local and global data access, and then again generalize these two nodes to one data read node independent of the access happening on the stack or the heap.

The *default taxonomy* is considering the data type of expression nodes, leading to the most relevant pattern being a specific data type, falsely indicating that a data type is responsible for an access violation. The *data flow taxonomy* correctly identifies the data access nodes, independent of their data type or global vs. local accesses as the pattern responsible.

Even in this example it could also be interesting to consider different variations of the *data flow taxonomy*. The presented taxonomy (see Figure 4.5) distinguishes data access as allocation, global and local accesses. An alternative would be to first distinguish between stack (local) and heap (global) acesses and later refine these into allocation, global and local in the next specialization.

The concept of taxonomies also enables the possibility of restricting the search space itself. When it is known that some specializations are irrelevant for mining, the search space can be restricted to be only evaluated in higher levels of the selected taxonomy. In a similar manner, the taxonomy can only be defined containing the nodes that are relevant for the patterns to be analyzed, e.g., only considering control- and data flow nodes but not expression nodes. This however also requires understanding if the order or structure of the ASTs is relevant for the resulting patterns.

4.3 Extending Patterns With Wildcards

A challenge in mining source code for patterns is that the specific behavior of a function may be influenced by multiple nodes. These nodes may be located in close proximity. Consider Listing 4.2 in the context of run-time performance. The two for loops and their respective conditions for looping over a structure result in $O(n^2)$ and are in close proximity to each other, making this pattern easily identifiable. When considering the same example in the context of data flow, the reason that the reading access to the variable $arr[j]$ succeeds, is because arr has been set on the stack, via the function argument. These nodes are not in close proximity to each other, and when several trees with successful data access would be mined it is likely that two separate patterns, one for function argument access, and one for reading access would be found with no connection between them. This issue has been identified in literature concerning function calls to API elements which build a usage pattern together [48].

Listing 4.2: Example of pattern locations using bubble sort. The *for→for* contained loop pattern is in lines 4 and 5 and the *read function argument -> arr[j]* pattern is in lines 1 and 6

```
 1  int[] bubbleSort(int arr[], int n)
 2  {
 3      int i, j;
 4      for (i = 0; i < n-1; i++) {
 5          for (j = 0; j < n-i-1; j++) {
 6              if (arr[j] > arr[j+1]) {
 7                  swap(&arr[j], &arr[j+1]);
 8              }
 9          }
10      }
11      return arr;
12  }
```

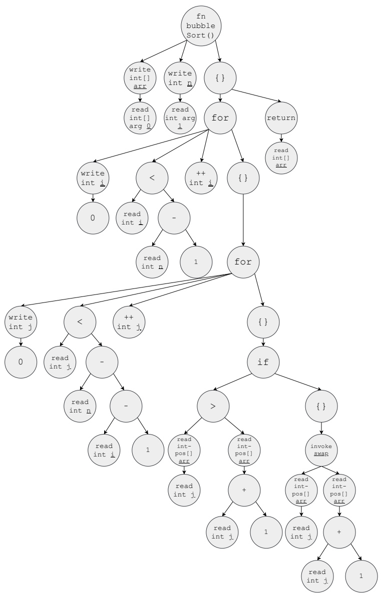


Figure 4.6: Bubble sort represented as AST.

To solve this issue, wildcards can be used to connect distinct patterns. Figure 4.7 shows a combined pattern of *(write)→(read)*, where the (★) wildcard represents that the *(write)* and *(read)* must be in the same AST and that the order of these patterns must be upheld, but where exactly they are located in the tree is not relevant. The pattern *(write)→(read)* itself causes additional issues though.

This example leads to the question of how patterns are considered. From the viewpoint that *(write)→(read)* will not produce a run-time exception, the pattern is correct. When an AST accesses a global variable, the corresponding *write* is not necessarily represented in the AST, thus creating the valid option of the pattern *read*. This issue can be circumvented with considering the taxonomy being mined and defining correct data access with two patterns *write-local→read-local* and *read-global*. Another option, that is outside the scope of this work, would be to add run-time information to the ASTs and record global variables which are available in the global scope.

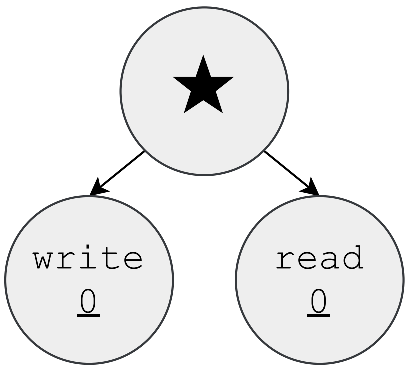


Figure 4.7: Use of the wildcard (★) in a pattern to define that in any given AST a read must have a corresponding *write* earlier in the AST.

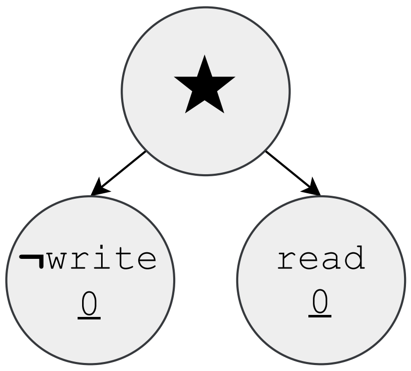


Figure 4.8: Use of the wildcard (¬) in a pattern to define a fault of omission. In an AST a read without a corresponding *write* earlier in the AST is a bug.

Definition 4.3.1 (⋆) *is a wildcard node that fixes the order of occurrence of its child nodes in an AST. It represents* $1..n$ *nodes of any structure in between.*

In the context of pattern mining, the wildcard pattern (⋆) as specified in Definition 4.3.1 increases the search space even more than the utilization of a taxonomy, as direct relations between nodes could simply be skipped. Thus, the (⋆) wildcard is optional during the mining process. Pattern mining in source code does not usually distinguish between that stem from a direct or indirect relationship [30]. The star wildcard however denotes explicitly any location where at least one node is in between nodes connected by the (⋆) wildcard, i.e. an exclusively indirect relationship.

In addition to the wildcard pattern (⋆) for representing $1..n$ another wildcard (•) is also considered. The (•) wildcard is always equivalent to the highest point in the used taxonomy, e.g., a node of any type, and thus requires no special consideration. It exists primarily for the purpose of maintaining patterns where the (⋆) wildcard is too generic, and the structure of how patterns are related matters, while the actual type of the nodes in the structure matters less.

Definition 4.3.2 (•) *is a wildcard that represents a node independent of a type. It defines a relevant structural aspect of an AST.*

This ties in with the concept of taxonomies not necessarily having to represent the entirety of a programming language's concepts, but rather just the subset relevant for mining. Nodes observed in a search space that do not occur in a given taxonomy can be generalized to (⋆) if the order in which the nodes occur is relevant, but the structure is not. Alternatively, if the structure is also important nodes not occurring in the taxonomy can be replaced with (•).

Another challenge in pattern mining is faults of omission, i.e. faults caused by the absence of code. When considering Figure 4.7 a fault of omission would be the read to a variable that has never been initialized, as shown in Figure 4.8. Such a pattern can be defined with a negation wildcard (¬). In the context of this work, the (¬) wildcard is not applied during the mining process, but rather during manual analysis of faults compared to faults of omission.

Definition 4.3.3 (¬) *is a wildcard that negates a node to identify if the absence of a given node is significant.*

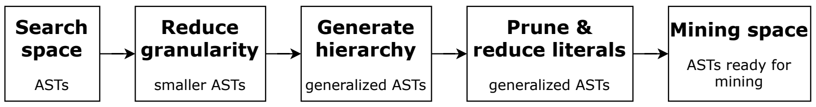


Figure 4.9: Pipeline for AST normalization before the mining begins. All shown steps are optional, but each ensures that the mining process produces more general patterns in a shorter time.

4.4 AST Normalization

When analyzing the pattern *write→read* (Figure 4.7) an immediate flaw can be noticed concerning the data flow. A *write* to variable x with a corresponding read from variable y does not guarantee that y will be initialized. This indicates the need for a normalization of ASTs. Nguyen and Nguyen [48] suggest that in addition to variable names, several special values such as *null*, the integer value *0* and empty string literals *""* should also be normalized instead of removing them completely. Thus, a need for a generalized normalization to improve the mining of patterns occurs for the following items in code:

granularity The chosen representation form is very fine-granular. This results in large ASTs and thus large search spaces. This caveat can be mitigated by pruning AST paths are not relevant in the context (e.g. pruning terminals, etc.) or combining them into larger structures (e.g. reducing large math-terms to a single expression).

taxonomy The taxonomy can be used to generalize unnecessary specializations of nodes. As it would be a performance impediment to do this repeatedly during mining, the ASTs can be normalized beforehand. In a similar manner, nodes not occurring in the taxonomy can be replaced with the (⋆) or (•) wildcards, depending on if the structure and indirect relationships matter or not.

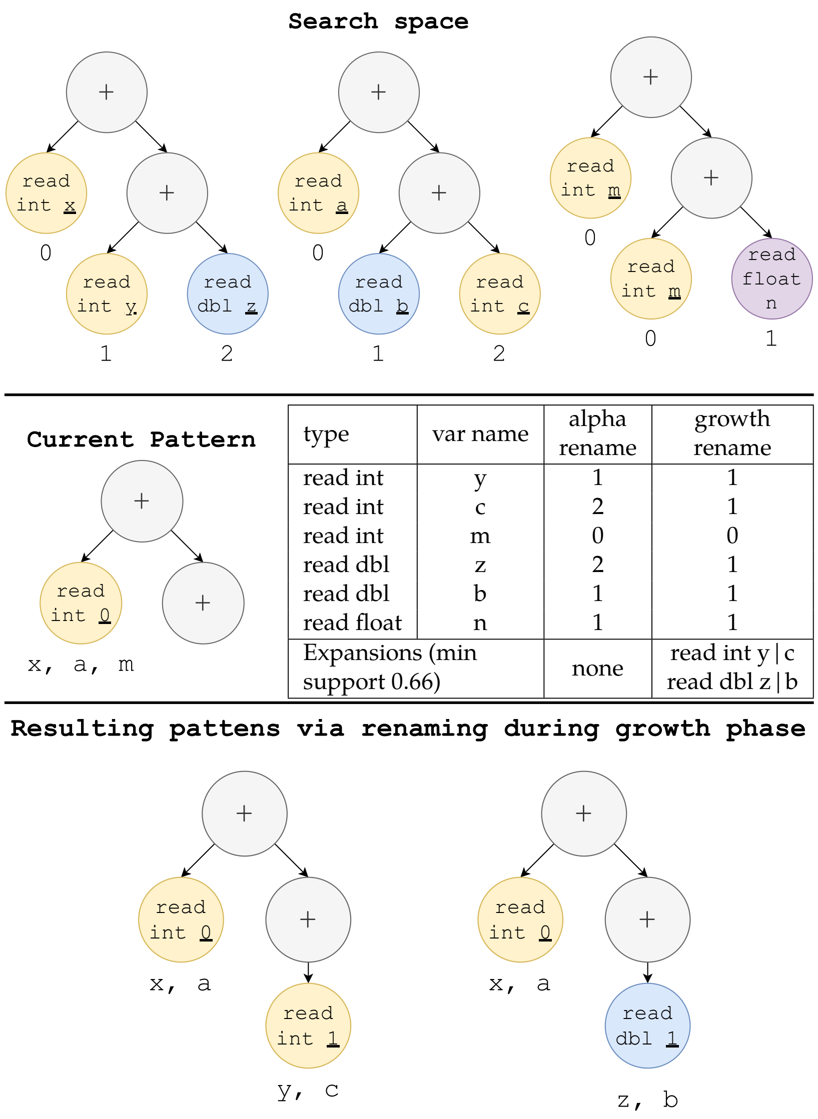
variables variable names must be normalized and kept in a pattern. This is a challenging task, as a pre-normalization of ASTs is not possible since this would lead to patterns not matching due to incoherent variable-naming between different ASTs. This normalization must be applied during the growth phase of the pattern mining algorithm.

literals that are interesting in the mining context, such as *null* or *0*, can be replaced by labels for each interesting literal, while uninteresting literals are simply removed.

function calls are challenging to be normalized as what is being called can change over different executions due to dynamic binding. What can be done, however, is a normalization of the available static information such as the function signature.

The above normalization steps can be categorized into modifications of the *tree structure*, the *type of node* and the *content of node*. These normalization steps have to be performed in that order, as the structure modifications remove nodes, and changing the type of a node may require a change of its contents. This results in a pipeline as shown in Figure 4.9 of normalization steps which are done in the defined order.

Concerning the normalization of the *tree structure*, special consideration needs to be taken towards the relationships to child-nodes that may not be merged into a more general node but remain their own node. These must remain in the original order of the ASTs related to each other.



type	var name	alpha rename	growth rename
read int	y	1	1
read int	c	2	1
read int	m	0	0
read dbl	z	2	1
read dbl	b	1	1
read float	n	1	1
Expansions (min support 0.66)		none	read int y|c read dbl z|b

Figure 4.10: Variable labelling with three different strategies. Considering the pattern in the middle left, the expansion table middle right becomes possible considering the search space at the top. When not renaming the variables, or preprocessing with alpha renaming, no expansion with a *minimum support threshold of 0.66* is possible. When renaming during the observation, two expansions remain possible, while still correctly pruning the third tree with *m* previously recorded as 0.

Modifying the *type of a node* only requires the consideration of its contents and relationships. As several nodes are combined into one type, e.g. a *write node* and a *read node* into a data-access node, the content needs to be merged. When for example, the *write node* defines the field in which the variable is stored as "slot" and the read node defines it as "varName", the fields should be named the same when either is generalized to *data access node* to strengthen found patterns. In some cases, this can be done automatically, for example when both classes have only one field relating to a variable, or when they are named the same, e.g. *while* and *if* both have the relationship *condition*. In other cases, a manual mapping of fields and relationships is required.

Concerning the *content of a node*, for variable names Nguyen and Nguyen [48] utilize alpha renaming (also called alpha conversion). This would rename an integer variable *x* into *var0_int*. Our work intentionally does not use alpha conversion for two reasons. The first is, that this type of renaming would prevent the advantages of the taxonomy, as variables of different data types could not be combined into the same pattern anymore. The second reason is, that when utilizing a pattern growth approach, the variables can't be numbered during a normalization phase.

To ensure that patterns uphold the data- and control-flow (for variable names and function signatures, respectively) this normalization alone has to happen during the growth phase, to ensure that variables already in the current pattern are upheld.

Figure 4.10 shows a pattern in the middle section that will be grown (extended by one node). From the given search space at the top, four nodes can be expanded from the lower (+) node. Two of them are of type *read int* and two are of type *read double*. Via accessing the variable names, or via a preprocessing through alpha renaming, even without using the data type in the alpha name, no expansion is possible that would keep at least 2 trees, i.e. a *minimum support threshold of 0.66*. However, when renaming during the growth phase, both the *read int* and the *read double* can be expanded, as neither variable has been observed before and can be assigned the value 1. The third tree in the search space, just like with alpha renaming, gets pruned from the pattern as m has been observed before and is assigned 0.

4.5 Encoding Abstract Syntax Trees

The chosen representation as a tree poses a limitation on the search space because of it's composition. As an acyclic graph with variable depth, it consists of nodes and relationships between the nodes. When conducting mining on an object-oriented representation of such an AST, memory becomes a bottleneck. ASTs consist of several hundred nodes, with 2^n components, and several hundreds or thousands of AST may be mined together. In a similar manner, due to the object allocations and continuous tree traversals, performance becomes an issue as well. Thus, an encoding serves to reduce the size of the AST structures, and allows efficient comparison operations.

To properly capture a pattern in any given granularity the following values must be captured in a pattern:

type The type of the node at the pattern's selected specialization/generalization level in the taxonomy.

values Non-pruned values such as variable and function name replacements or relevant selected literals (0, null, ...)

structure How the nodes are connected to form a tree.

Literature rarely mentions the chosen encoding often (3 out of 34 publications on mining, see Section 7.1). [32] only mentions that the encoding can reduce the size of the mined data. Those that do go into detail about simplifying a representation for mining, choose a string encoding. For example [30] chooses a depth-first string encoding via the labels of trees. Every time a child returns to the parent node a $ is inserted into the string. [40] encodes lines of code to hash values for further processing.

The following sections present an encoding of trees exclusively via integer types, reducing the structure and type down to the bit-level.

Taxonomy Encoding

Algorithm 3: Taxonomy to Bitmask Encoding

```
Data: taxonomy
Data: root
/* Initialization                                              */
1  layer ← 0;
2  bitSize ← 0;
3  typesInLayer ← { root };
4  taxonomyToBitmaskMap ← (root → 0) };
5  bitmaskToTaxonomyMap ← { (0 → root) };
6  layerMap ← {};
/* descend taxonomy layers                                     */
7  while typesInLayer ≠ ∅ do
       /* find required amount of bits by largest group of
          children                                             */
8      size ← max(typesInLayer.map(type →
         size(taxonomy.childrenOf(type))));
9      bitSize ← bitSize + ceil($\sqrt{size}$);
10     descendingTypes ← typesInLayer;
11     typesInLayer ← {};
       /* Mask all children                                    */
12     foreach type ∈ descendingTypes do
           /* Skip single size children                        */
13         while size(taxonomy.childrenOf(type)) = 1 do
14             type ← taxonomy.childrenOf(type)[0];
15         end
16         bitCounter ← 0;
17         parentMask ← taxonomyToBitmaskMap(type);
           /* Assign 0..n to each child in the layer           */
18         foreach child ∈ taxonomy.childrenOf(type) do
19             typesInLayer.add(child);
               /* Bit operation combining the parent bitmask with
                  the child bitmask                            */
20             childMask ← parentMask & (bitCounter << 64− bitSize);
                taxonomyToBitmaskMap.add(child → childMask);
                bitmaskToTaxonomyMap.add(childMask → child);
                bitCounter ← bitCounter + 1;
21         end
22     end
       /* Add size of layer to map                             */
23     layerMap.add(layer → bitSize);
24     layer ← layer + 1;
25 end
Result: taxonomyToBitmaskMap
Result: bitmaskToTaxonomyMap
Result: layerMap
```

The taxonomy in which the nodes are parsed are encoded to provide a bitmask for the type of a node. The approach to encoding is targeted towards identifying generalizations or specializations of a given type. It allows fast comparison if a type generalizes another type, which is relevant when filtering or comparing patterns in the mining process.

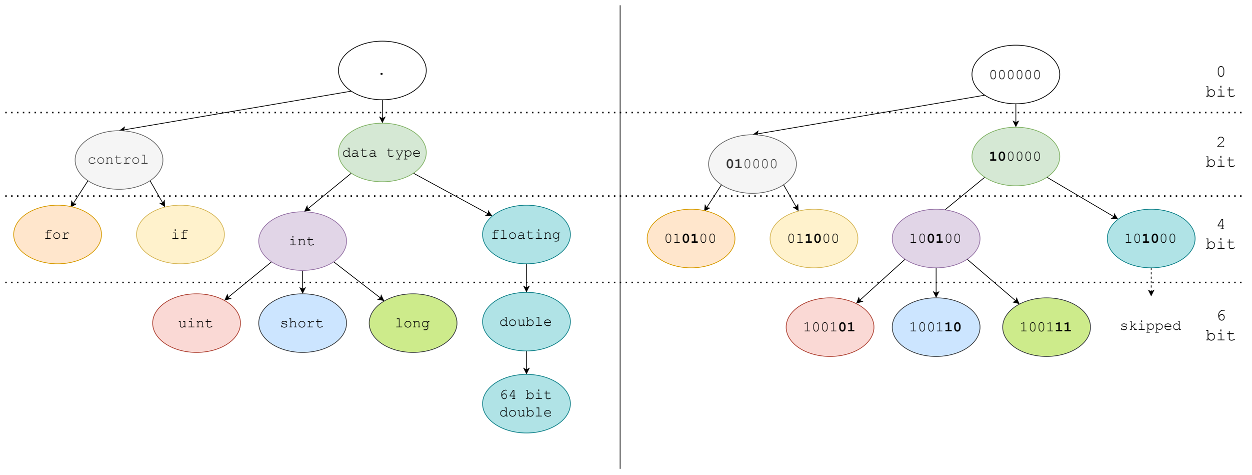


Figure 4.11: Encoding of the taxonomy (left) into bits (right - 58 "0" to the right omitted). Each layer only requires two bits for encoding. Irrelevant single item layers are omitted.

Algorithm 3 is used to encode a taxonomy to a bit-encoding. The layers of the hierarchy are reduced to bitmasks, identifying which bit identifies which layer. Bits further to the left identify higher layers (more general) in the hierarchy, bits further to the right identify lower layers. For each layer inside the bitmask, each type is numbered from 0 to n. This number is then combined with the value of the current parent to generate a unique identity of minimal size for each type in the taxonomy. Individuals with only one child in the taxonomy are skipped, as they do not provide any relevant information. For example considering *IntRead*→*IntReadLocal* and *IntRead* has no additional child types, any pattern producing *IntRead* can only be specialized to *IntReadLocal* and can thus be considered identical. As this generates no value, but increases the search space and produces irrelevant patterns, the encoding skips single-sized child nodes. An example taxonomy and it's encoded equivalents are shown in Figure 4.11.

Algorithm 4: Evaluate Generalization of Types

```
Data: bitMaskSpecialized
Data: bitMaskGeneralized
Data: taxonomyEncoded
/* Get mask size from layerMap in corresponding taxonomy
   encoding                                              */
1 size ← taxonomyEncoded.bitMaskLength(bitMaskGeneralized);
/* Bitshift and compare                                  */
2 if bitMaskGeneralized >>> (64-size) = bitMaskSpecialized >>> (64-size)
   then
3     return True
4 else
5     return False
6 end
```

To ensure that the bitmask allows generalization and specialization it is *left aligned*, meaning that a resulting bitmask can be read from left (general) to right (specialized). This is the reason for the bit shift operation in line 20 of Algorithm 3. This also enables fast checking if a given type is a generalization or specialization of another type via bit shifting of the

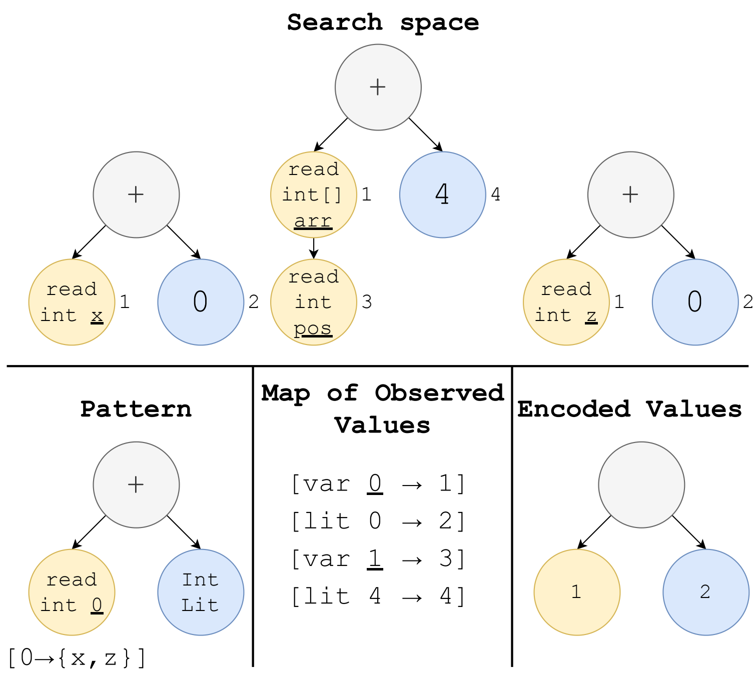


Figure 4.12: Encoding of the values in a node. The top shows the search space, while the bottom shows from left to right - the pattern, the map of observed values and the encoded values in the pattern per node. The list of observed variables mapping to the original variable names per tree is preserved in the encoding. The encoding shows only the left and right AST from the search space, as the middle one does not match (read int[] is not equal to read int).

size of the bit-layer as defined in Algorithm 4. The encoding is optimized towards size re-using the same bits in lower layers, as the upper-layers already declare the difference. If the encoding would not reuse bits (moving the bitCounter from line 16 before line 12 in Algorithm 3) a simple bitMaskSpecialized|bitMaskGeneralized == bitMaskSpecialized would be enough to validate generalization.

Value Encoding

Value encoding happens after the full AST normalization, including the normalization of variable names during the mining process, has been conducted. If a node has no content (field values such as IntLiteral.value) the value "0" is assigned as label to represent the node content. Otherwise all values remaining in the AST (see Figure 4.1) are sorted by the name of the field, and then combined into one hash. During the mining process this hash is stored in a map of (hash → long) mapping it to observed content types. Only the long value is stored in the encoded AST. All variable names that have been observed in the AST are stored in the encoded AST as well to enable the variable renaming during pattern growth. The mapping can be seen in Figure 4.12.

For analysis purposes, the values may be required to be displayed at a later stage. As they are not preserved in the encoding, this can be done via loading the nodes corresponding to the pattern and performing the selected normalization steps again. For the same reason, there is no need to store variable names in the specific nodes, but only which variable names have been previously observed. This allows us to combine variables, and other values in a node that were not removed during normalization, into one single label.

Structure Encoding

Algorithm 5: Depth First Encode AST

```
Data: ast
1  bitPos ← 63;
2  node ← ast.root;
3  structureBitmap ← 0;
4  while node ≠ ∅ do
5      if node.hasUnvisitedChildren() then
           /* Descend, as all positions are 0 at the start, no
              write is needed                                   */
6          node = node.nextUnvisitedChild();
7          bitPos ← bitPos - 1;
8      else
           /* Write 1 to close the subtree and ascend           */
9          structureBitmap ← structureBitmap +1 << bitPos;
10         bitPos ← bitPos - 1;
11         node = node.parent;
12     end
13 end
Result: structureBitmap
```

The structure of a given AST is encoded independently of the content of its nodes. This enables the creation of a structural encoding that uses only two bits per node in a tree. The encoding utilizes the bit 0 to denote the beginning of a subtree, and 1 to denote the closing of a subtree. As every AST must have at least one node the root node is not encoded. This enables encoding any given AST with a length ≤ 33 nodes into a single 64 bit data type. ASTs are encoded in a depth first manner as defined in Algorithm 5. Figure 4.13 shows the result of running the algorithm on a given pattern.

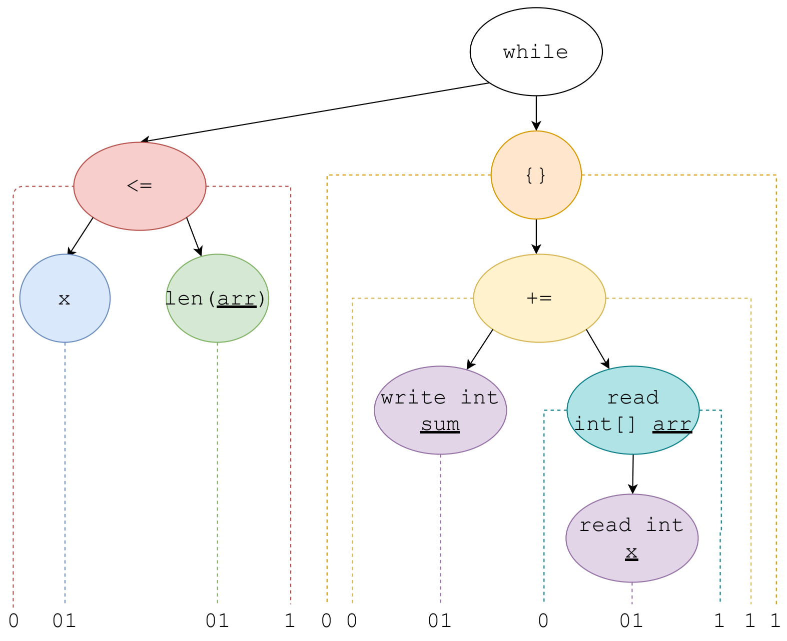
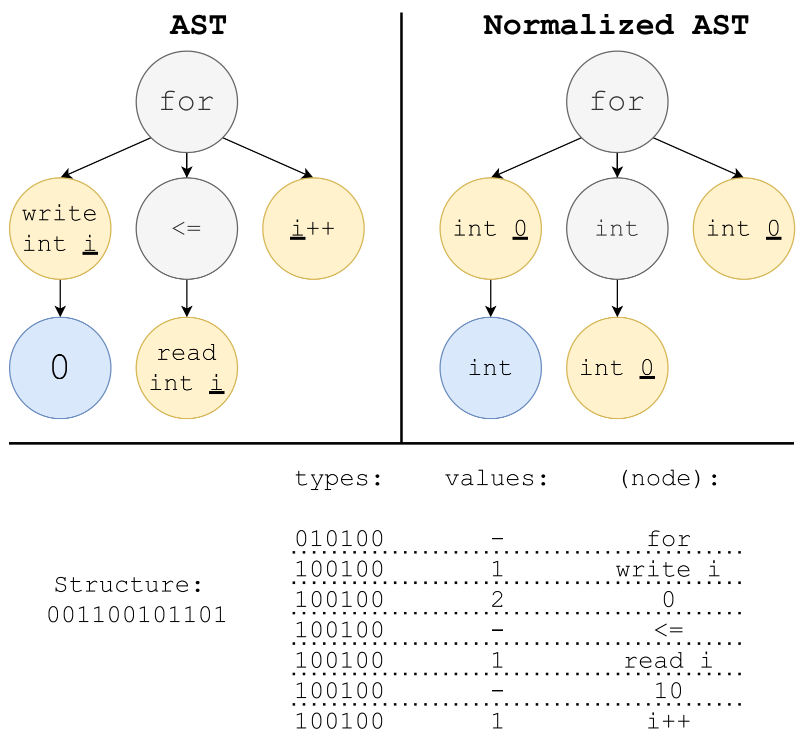


Figure 4.13: The AST node relationships are recorded into opening 0 and closing 1 bits. Corresponding bits are shown at the bottom.



```
                     types:     values:     (node):

                     010100        -          for
Structure:           100100        1        write i
001100101101         100100        2           0
                     100100        -          <=
                     100100        1        read i
                     100100        -          10
                     100100        1          i++
```

Figure 4.14: Full encoding of a given pattern on the left. The right shows the structure, type and value encodings as used in the pattern mining algorithms.

AST Encoding

The previous three sections explained how the different parts of the tree, types, values and structure are encoded into integer values. The combination of these three encodings results in a single encoding for a given AST or pattern. Figure 4.14 shows an example AST of this encoding utilizing the previously defined taxonomy encoding (see Figure 4.11) and a map of observations (see Figure 4.12). The top of the figure shows the sample AST, the bottom shows the encoding consisting of structure, types and values, with the node only as reference for comparison with the AST. The final result is a lean encoding, enabling efficient comparison and modification algorithms for the analysis and growth of a given pattern.

It is worth noting that the encoding as described has a theoretical limitation with the chosen 64 bit representation, as evident in Algorithm 3 and Algorithm 5. For brevity, the solution to this limitation has been omitted from the descriptions and algorithms. The encoding sizes can be dynamically raised by checking if the structures fit into a single 64 bit value and otherwise expanding the representation to an array of 64 bit values. This is unlikely to happen in the taxonomy encoding, as the taxonomy would have to either exceed a depth of 32 with 2 types per layer, or have thousands of nodes more evenly distributed. For example, the MiniC test language (see Chapter 9) has 357 types in its full taxonomy and uses only 19 bits of encoding space. Due to the granularity it is more likely that a pattern, or an encoded AST exceeds 33 nodes as can be seen in Figure 4.6 which is a simplified version of bubble sort with fewer nodes than the real tree produced by an interpreter.

Operations on the Encoding

The finished encoding allows several highly efficient operations to compare patterns:

equals Equivalence checking in the best case has a performance of $O(1)$ if the structure of the pattern does not match. In all other cases, the worst possible performance is $O(n)$, whereby n is the amount of nodes in the pattern.

contains performs slightly worse than equivalence checking, as a smaller tree's occurrence must be found in the structure. This requires iterating through the bit positions of found matches.

generalizes/specializes is a combination of equals and the bitmask comparison (see Algorithm 4), and has the same performance as equals.

contains generalization/specialization can be done with the same computational complexity as a regular contains operation.

The above operations find its use primarily in the cases of detecting patterns in later stages, such as the analysis of mined patterns. It is also relevant for identifying a (sub)AST generated with Genetic Improvement (GI), parsed from source code, or produced by a step in the interpretation or compilation process.

The primary operation that is important for the mining process is the *growth* of a pattern, i.e. adding one single node to the structure. The encoding also enables this with $O(1)$ in most cases, as pattern growth follows a rightmost-expansion process, i.e., in most cases a node will be appended at the end of a pattern. However, as patterns have multiple growth directions that must be evaluated they are copied, meaning that pattern growth always results in a performance of $O(n)$.

4.6 Cluster Pattern Mining

The previous sections explained the foundations, *representation* (Chapter 4.1), *taxonomies* (Chapter 4.2), *wildcards* (Chapter 4.3), *normalization* (Chapter 4.4) and *encoding* (Chapter 4.5) used in the mining approach. These parts are now combined into one algorithm for mining.

In the context of compilers and interpreters, both *significant pattern mining*, i.e. mining a search space for frequently occurring patterns, and *discriminative pattern mining*, i.e. mining a search space for patterns that show a discriminative difference between two groups (positive and negative) are important. This work expands the concept of discriminative pattern mining by introducing a mining algorithm that can differentiate between any given amount of groups. This is necessary to answer the research question on the relationship between patterns and their properties[RQ1], as non-functional properties often have to be considered in a range other than *positive* and *negative*. For example, the analysis of what impact data types have on run-time performance, would require a cluster per data type under analysis (e.g. *int*, *double*, and *string*). Thus, the concept of *cluster pattern mining* and a novel algorithm for it is introduced. The name has been chosen as the targets for mining are likely to build clusters in one or more axes of the given search space over all ASTs.

RQ1: How can recurring patterns be identified that impact or improve a functional or Non-Functional Property?

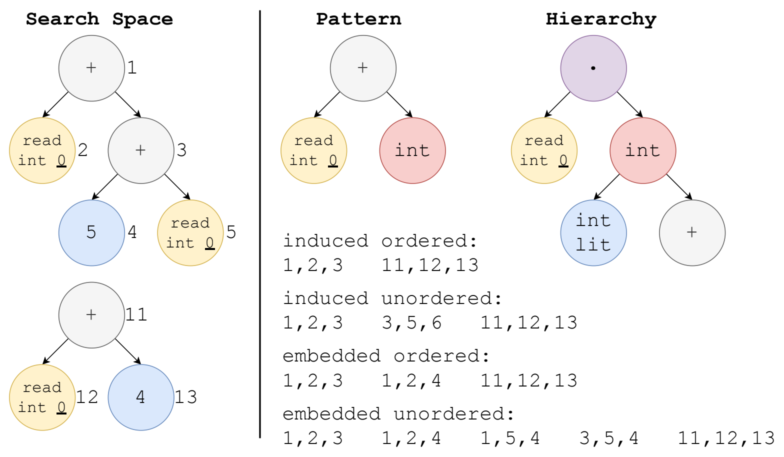


Figure 4.15: Relationship between a pattern and ASTs that it has been mined from. The locations are shown according to to all combinations of *ordered*, *unordered*, *induced* and *embedded*

Cluster pattern mining has nothing to do with attempting to find out in which cluster an AST lies (cluster analysis), but rather to mine patterns that are discriminative in already known clusters.

As *discriminative pattern mining* (discriminating between two known clusters) is just a subset of *cluster pattern mining* no separate algorithm for it is needed. Similarly *significant pattern mining*, can be viewed as a specialization containing only one (unnamed) cluster. Thus, only one algorithm is needed to cover all three concepts. What is different, however, are the metrics that can be applied depending on how many clusters are utilized in the mining.

Relationships Between Patterns and ASTs

To understand how mining algorithms work, the relationships between patterns and ASTs must be defined more clearly. The algorithm presented in this section is a pattern growth algorithm for *induced or embedded, ordered* patterns, with the ability to find all *locations* of a pattern over multiple trees.

In the context of pattern mining, patterns are considered to be in an AST in four different ways. *Ordered* vs. *unordered* combined with *induced* vs. *embedded*. Figure 4.15 shows a tree, and a pattern with its *locations* according to all four different ways. *Ordered* means that the original order of child relationships must be preserved, whereas *unordered* means that the order of child relationships is irrelevant. *Induced* means that the nodes must be directly connected, whereas *embedded* means that $1..n$ child relationships can be used to identify a pattern. Different algorithms will be more or less efficient depending on the properties chosen for mining, or may not be able to support them at all.

In the context of source code, it is relevant how statements and their components are related to each other. The structure of the code is important, and at the very least the fact if a node is a direct descendant or anywhere below in the hierarchy should be taken into account. Thus, *induced* and *embedded* patterns are considered via a clear separation of via the ($\star$) wildcard. In most cases, source code should also be considered in an *ordered* fashion. Sometimes *unordered* may be preferred, as the order of source code might not be relevant as long as the data flow remains the same. This depends on the NFP being mined, as the order of nodes has an impact on some of them. For example, run-time performance changes with the order of statements, as utilized in the code motion optimization [4].

Metrics for Mining

Both *significant pattern mining* as well as *discriminative pattern mining* have well-known metrics that are utilized to find relevant patterns, and enable the ranking of found patterns. *Significant pattern mining* considers *support thresholds* i.e. limiting the minimal or maximal occurrence of patterns. Lucia et al. [85] give a comprehensive overview of metrics used in fault localization that can be used when comparing two classes for *discriminative pattern mining*.

Unfortunately, the known metrics cannot be utilized in the given context of cluster pattern mining. Discriminative metrics can only differentiate between two clusters and must be extended to consider n clusters. All metrics, significant and discriminative, also need to be extended to consider the concepts of *taxonomies* and *wildcards*. Thus, in this context, the definition of significant and discriminative must be adapted.

Pattern Size

Redundancy in patterns, i.e., patterns that grow around a core pattern and have the same significance or discriminative value, is an issue [35, 37]. Cheng et al. [37] propose that smaller patterns should be preferred, as they hint towards the core issue being analyzed. In contrast to them, Hanam, Brito, and Mesbah [49] suggest that the context in which a pattern is embedded is of great help to understand the pattern, and other work attempts to find larger meaningful patterns [44], even going so far as to merge overlapping patterns into larger ones [40].

In the context of being significant or discriminative, both viewpoints lead to meaningful definitions. A *compact pattern* as defined in Definition 4.6.1 can be utilized to reduce multiple branches around the same core pattern. This may be relevant when identifying the exact location at which an exception occurs. The definition always considers the largest pattern, that other patterns branch away from. For example, if there is a pattern A, that is contained in pattern B and C, pattern A would be compact. However, if another pattern D exists, containing pattern A, B and C, A is not compact, but pattern D is compact.

Definition 4.6.1 *A pattern is* ***compact*** *when there are multiple patterns containing it, but no other pattern exists that contains them.*

In contrast, a compact pattern can have many *closed patterns* (Definition 4.6.2) growing around it. These closed patterns may give more information towards the context in which a pattern exists, and with that possibly the location of the bug causing the exception. If for example a pattern A exists that is compact, and pattern A is contained in patterns B and C, patterns B and C are closed, so long as there is no pattern D containing either of them.

Definition 4.6.2 *A pattern is* ***closed*** *when there is no larger pattern containing it.*

Significance Metrics

A pattern can be considered significant (see Definition 4.6.3) when it adheres to a given significance metric. Considering the *taxonomy*, and the (•) wildcard a pattern is only significant, if there is no pattern that is equivalent in structure, but has nodes that are more specialized. For example, consider two patterns A) (int write)→(int read), and B) (write)→(int read). Both patterns are structurally similar, and since (write) is more general than (int write), pattern B) will contain all ASTs that are also contained in pattern A). If there are no ASTs that conduct a (write) of a different type, both patterns are equivalent considering the frequency the pattern occurs in. This means that according to Definition 4.6.3, only the more specialized pattern A) is significant. If there are ASTs with a (write) of a different data type, both pattern A and pattern B would be significant. This allows the mining process to find the most specialized patterns, while also finding more general patterns that represent more sub-ASTs, which can identify more generally applicable patterns.

Definition 4.6.3 *A pattern is* ***significant*** *when it lies within the* thresholds *of a given significance metric, and there is* no equivalent specialization *of that pattern with the same value assigned by the significance metric.*

The significance metric can also be used to determine the strength of a pattern, i.e. if it is more significant or less significant than any other pattern. In most cases, *more significant* means that the metric is higher, e.g., the pattern occurring in more trees. In some cases, however, for example for outlier detection, the ranking may be determined by less frequently occurring patterns.

The following significance metrics are relevant for this work:

support threshold Equation 4.1 - the minimum and maximum support threshold is defined by the amount of *ASTs* a pattern occurs in, over all ASTs in the search space. This is important to find patterns that occur in multiple different ASTs. Let *searchSpace* be the set of ASTs in the search space:

$$\text{support(pattern)} = \frac{\left|\{\text{searchSpace}|\text{pattern} \in \text{searchSpace}\}\right|}{\left|\text{searchSpace}\right|} \tag{4.1}$$

frequency threshold Equation 4.2 - the minimum and maximum frequency threshold is defined by the amount of occurrences of a pattern overall. Unlike the support threshold it considers multiple occurrences in the same *ASTs* to play a different role. This can be an important indicator if a pattern can be used for optimization, or to detect code duplication in AST. To ensure that the frequency is ranged between 0 and 1, it is via the amount of nodes in the search space. Let *occurences(pattern, AST)* be the amount of occurences of a pattern in an AST.

$$\text{frequency(pattern)} = \frac{\sum_{\text{AST} \in \text{searchSpace}} \text{occurences(pattern, AST)}}{\left| \{ \bigcup_{AST \in \text{searchSpace}} \text{node} \in \text{AST} \} \right|} \tag{4.2}$$

Discriminative Metrics

Similar to significance, a pattern can be considered discriminative (see Definition 4.6.4) when it adheres to a given discriminative metric, and there is no pattern that is equivalent in structure, but has nodes that are more specialized. Most metrics as defined in literature [39, 85] could be adapted towards supporting more than two groups.

Definition 4.6.4 *A pattern is* ***discriminative*** *when it lies within the* discriminative thresholds *of a given discriminative metric, and there is* no equivalent specialization *of that pattern with the same frequency.*

In the context of analyzing the difference between clusters, the most interesting rankings are those that either ensure that a pattern is most distinctive in one cluster, or that a pattern generates a distinction over all clusters. Both of these metrics are the generalization of the *Contrast* metric to multiple groups.

average contrast Equation 4.3 - calculates the support metric for each cluster. The average contrast between all classes is calculated from that. Let *clusters* be the set of clusters. Let *support* be Equation 4.1.

$$\text{avgContrast(clusters)} = \frac{\sum_{a,b \in clusters, a \neq b} |support(a) - support(b)|}{|clusters|} \tag{4.3}$$

maximum contrast Equation 4.5 - is also based on the support metric. It calculates the largest distance of all clusters to their closest neighbours.

$$\begin{aligned} \text{minContrast(a)} = & \min\{|\text{support(a)} - \text{support(b)}| \\ & \text{b} \in \text{clusters}, \text{a} \neq \text{b}\} \end{aligned} \tag{4.4}$$

$$\text{maxContrast(clusters)} = \max_{\text{a} \in \text{clusters}} \{\text{minContrast(a)}\} \tag{4.5}$$

Independent Growth of Ordered Relationships (IGOR) Algorithm

Independent Growth of Ordered Relationships (IGOR) enables mining patterns in a search space of multiple ASTs with ordered relationships. It keeps track of all locations of a mined pattern, enabling an independent growth of each and every pattern within a provided search space. This means that the algorithm supports parallelization, as well as distributed execution over multiple machines, and pausing or restarting the mining approach at any given time.

The core approach of IGOR is a pattern growth process. This means that it identifies all patterns of size 1, and from this point on grows each pattern that satisfies one or more selected metrics, according to a range (minimum, or maximum) or alternatively according to a top-n approach of the metric. This guarantee is upheld via a rightmost-growth approach, meaning that patterns will be grown only to the right of where the pattern was grown previously.

The following paragraphs explain how IGOR works, by explaining its three core phases. Algorithm 6 is responsible for pre-processing the entire search space of ASTs, and transforming it into patterns of size 1. Algorithm 7 then explains the growth phase, in which all possible extensions of a pattern are analyzed, and conducted if the pattern is *significant* or *discriminative* which is analyzed in Algorithm 8. The growth phase happens iteratively, always growing patterns by one node until no pattern can be grown anymore.

Algorithm 6: Independent Growth of Ordered Relationships Algorithm Initialization Phase

```
Data: taxonomy
Data: searchSpace
/* Initialization                                              */
1  patterns ⟵ {};
2  growthMap ⟵ {};
/* Find all size 1 patterns                                    */
3  foreach ast ∈ searchSpace do
       /* Sort by user defined function                        */
4      orderedRels ⟵ sort(ast.getRelationships());
5      foreach node ∈ ast do
6          nodeNorm ⟵ normalize(node);
7          foreach taxonomyNode ∈ taxonomy.hierarchy(nodeNorm) do
8              pattern ⟵ createPattern(taxonomyNode,
                 orderedRels.withParent(taxonomyNode));
9              growthMap.add(taxonomyNode.id ⟶ pattern);
10             if pattern ∈ patterns then
                   /* When a pattern was already found in another
                      tree, add all locations to the already
                      observed one                             */
11                 patterns.get(pattern).addLocations(pattern);
12             else
                   /* When the pattern is new, it is added to the
                      list                                     */
13                 patterns.add(pattern);
14             end
15         end
16     end
17 end
/* prune size 1 patterns                                       */
18 foreach pattern ∈ patterns do
19     if ¬metric.satisfied(pattern) then
20         patterns.remove(pattern);
21     end
22 end
```

Algorithm 6 first transforms the search space of ASTs into all patterns of size 1. Each pattern has a map, in which every node that matches the pattern, points towards all of its child nodes. This *growth map* is used during the later growth phase. Figure 4.16 shows an example search space, and the patterns resulting from Algorithm 6. Line 4 of the algorithm relates to an unspecified *sort* operation for the relationships in the AST. This sort will create a map of a parent node to a list of all child nodes in a specified order. For a compiler or interpreter, the order will be the natural order of a construct. For example, *if* $\longrightarrow$ *(condition, then, else)* or *block* $\longrightarrow$ *(statement 1, statement 2, ...)*. Alternatively, the order can be via the relationship names, or the order of the AST nodes. The algorithm itself is agnostic to any specific tree, as long as the order is specified. Line 6 refers to the normalization as described in Section 4.4. Line 7 refers to the taxonomy in Section 4.2. Depending on the generalization level, a pattern can be grown from many more locations than its specializations. This makes it necessary to mine all required generalizations until the end of the process, unless they have the exact same locations (see Definition 4.6.3 and Definition 4.6.4). Line 8 transforms the given node into an encoded pattern of size 1 (see Section 4.5). How locations are tracked and how additional locations are added in line 11 is discussed in the next section.

The first time patterns can be pruned according to metrics (see Section 4.6) is after the entire search space has been evaluated (Lines 18-22), as all trees in all clusters must be considered for the metric.

Algorithm 7: Independent Growth of Ordered Relationships Algorithm Growth Phase

```
   /* From this point onwards patterns can be grown
      individually                                                */
23 foreach pattern ∈ patterns do
       /* Iterate through pattern and grow all possible
          positions                                               */
24     foreach position ∈ pattern do
25         growthOpportunities = pattern.getGrowthOpportunities(pos);
26         combinedOpportunities ← {}; foreach gOpportunity ∈
            growthOpportunities do
27             opportunity ← growthMap(gOpportunity);
               /* combine equal extensions for new pattern        */
28             if opportunity ∈ growthOpportunities then
29                 combinedOpportunities.get(opportunity).
                     addLocations(opportunity);
30             else
31                 combinedOpportunities.add(opportunity);
32             end
33         end
           /* Expand pattern in all directions that satisfy the
              metric                                              */
34         foreach combinedOpportunity in combinedOpportunities do
35             if metric.satisifed(pattern, combinedOpportunity) then
36                 grownPattern ← pattern.grow(combinedOpportunity);
37                 patterns.add(grownPattern);
38             end
39         end
40     end
41 end
```

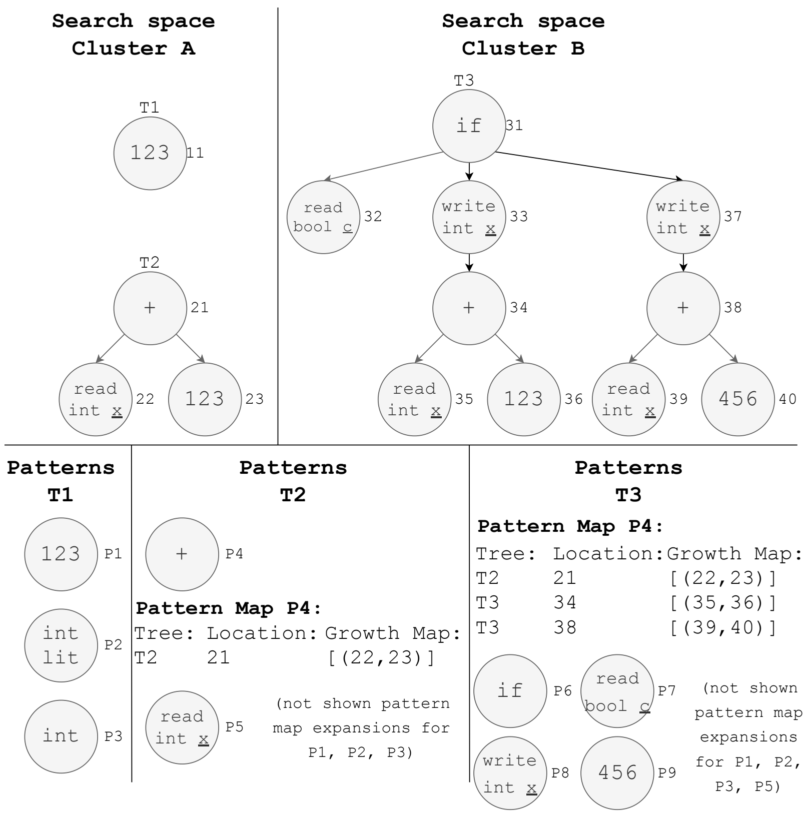
Starting with the growth phase of the algorithm (see Algorithm 7) it can be executed in parallel or in a distributed way without any form of synchronization. The reason for this, is that the algorithm guarantees that every individual pattern produced will be unique and evaluated only once. The only exception where a synchronization is necessary, is a top-n ranking of the metrics, e.g., only when mining and growing the top n relevant patterns. This requires a synchronization of the currently evaluated ranking. Figure 4.17 shows an example for how one pattern is grown.

From Line 25 in Algorithm 7 onwards, every position of the pattern is iterated and checked for growth opportunities. This is necessary, as a rightmost growth does not guarantee that only the last position in a pattern has growth opportunities. However, the next growth must only extend to the right of the newly injected position. Lines 26-34 are essentially repeating the size 1 growth phase for the currently evaluated pattern. They group and combine extensions that will lead to n locations for one new pattern. This also enables metrics to utilize these values before the actual growth happens, guaranteeing that only relevant patterns will be grown. The growth operation from Line 37 is discussed in the final subsection of this algorithm.

Algorithm 8: Independent Growth of Ordered Relationships Algorithm Pattern Reduction Phase

```
42 foreach pattern ∈ patterns do
43     foreach otherPattern ∈ (patterns \ pattern) do
           /* Satisfy significant and discriminative (breaks
              omitted)                                          */
44         if otherPattern.generalizes(pattern) ∧ metric(pattern) =
             metric(otherPattern) then
45             patterns.remove(otherPattern);
46         end
           /* Satisfy closed or compact                         */
47         if otherPattern.containsAndGeneralizes(pattern) ∧ metric(pattern) =
             metric(otherPattern) then
48             if prune = compact then
49                 patters.remove(otherPattern);
50             end
51             if prune = closed then
52                 patters.remove(pattern);
53             end
54         end
55     end
56 end
```

The first two parts of IGOR provide the essential initialization phase, and pattern growth, which is extending only relevant patterns. These patterns do not yet satisfy the definition of *significant* or *discriminative*. The final part Algorithm 8 deals with this. These conditions can actually be satisfied earlier (only the most specialized version of a pattern needs to be grown) as well as the *closed* definition, can be satisfied while adding a grown pattern in line 38 of Algorithm 7, but are discussed here for clarity. The only condition that can only be satisfied after evaluating the



```
Pattern Map P4:
Tree: Location:Growth Map:
T2    21        [(22,23)]
```

```
Pattern Map P4:
Tree: Location:Growth Map:
T2    21        [(22,23)]
T3    34        [(35,36)]
T3    38        [(39,40)]
```

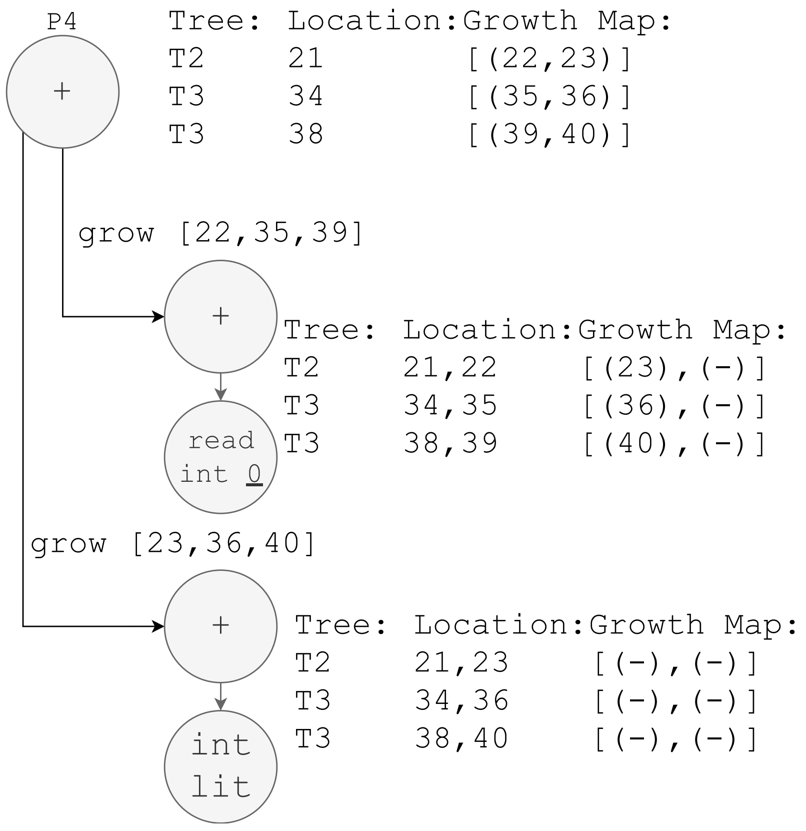
Figure 4.16: Tracking of locations in patterns. The top shows the search space. The bottom shows all generated patterns, in the order they are generated by Algorithm 6. To the left, all patterns from T1 of size 1 are shown. Then T2 (middle), creates the new (+) and (read int x) patterns. (123) is not generated as a new pattern, as it already exists from T1. Finally, all patterns from T3 are added (right). The pattern map covers all locations of all ASTs they are in. The growth map contains all not-yet explored relations to child nodes from the respective location and position. An example of how the pattern map and growth map are extended is shown by the pattern map of P4 (middle), being extended after analyzing T3 (right).

entire search space is *compact*, as patterns that are *closed* can grow into larger patterns that are *compact* again.

Tracking Node Locations

The locations of a pattern in ASTs are important for mining. As a pattern can occur multiple times in the same tree, and will be able to grow in different ways, all existing locations in all ASTs in the search space must be observed during the mining process. This requires an extension of the encoding to also consider all locations where a pattern is found, and all possible options for it to continue growing.

Figure 4.16 shows all patterns in a corresponding search space after they have been created as size 1 patterns via Algorithm 6. THe bottom left shows all patterns for the first AST in the search space (see Algorithm 6 line 8), and after a location has been added to it from the second AST in the search space (see Algorithm 6 line 11). It also tracks generalizations and specializations of other patterns. This can be used to prune patterns according to the *significance metric*. The trees are duplicated with the locations, to clearly identify which location exists in which tree. This allows evaluation of the existence in trees as well as the frequency of a pattern. The tree identities can also be used by the discriminative metrics to identify which cluster a tree is in. The growth map is a multi-dimensional array, relating for each location and position in a location which nodes can be grown from it.



Tree:	Location:	Growth Map:
T2	21	[(22,23)]
T3	34	[(35,36)]
T3	38	[(39,40)]

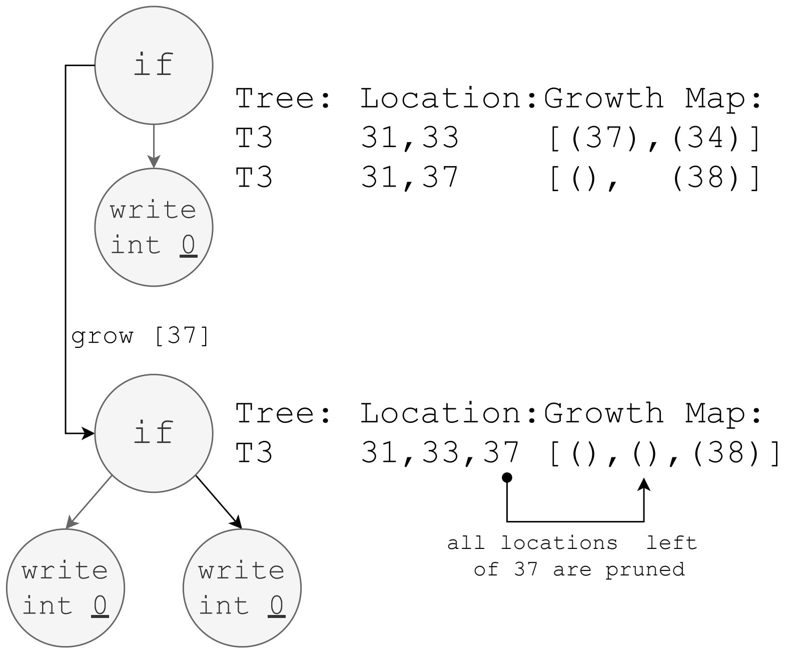
Tree:	Location:	Growth Map:
T2	21,22	[(23),(-)]
T3	34,35	[(36),(-)]
T3	38,39	[(40),(-)]

Tree:	Location:	Growth Map:
T2	21,23	[(-),(-)]
T3	34,36	[(-),(-)]
T3	38,40	[(-),(-)]

Figure 4.17: The pattern from Figure 4.16 is grown via both available opportunities. The upper pattern can still be grown, as the rightmost-growth approach leaves an opportunity to grow the leaf node to the right. The lower pattern cannot be grown anymore.

Growing a Pattern

From the locations, the opportunities for growth are returned per position (see line 25 in Algorithm 7). Considering the pattern after T2 (see Figure 4.16), the growth opportunities would be returned as (22, 23), (35,36), and (39,40). Algorithm 7 will cluster these three into only two growth opportunities (ignoring the taxonomy for this example) *(read int x)* with



Tree:	Location:	Growth Map:
T3	31,33	[(37),(34)]
T3	31,37	[(), (38)]

Tree:	Location:	Growth Map:
T3	31,33,37	[(),(),(38)]

Figure 4.18: A pattern loses a location during growth if the larger pattern can't extend to that position anymore. The growth opportunities to any nodes left in the hierarchy are pruned.

the locations [22, 35, and 39], and *(int lit)* with the locations [23, 39, 40].

The actual growing algorithm applies these opportunities. The encoding is expanded with a "01" at the correct location in the tree-structure, and the corresponding node is injected with the hierarchy and value encodings. The pattern locations are extended via a rightmost growth approach, meaning that only locations sorted after the injected location are considered as new growth points. Figure 4.17 shows the original pattern (+) and both new pattern expansions with their corresponding search spaces. In this particular example, no new growth points are added as all new nodes are leafs in their tree, but any new location will add all relationships of the injected nodes as new growth opportunities. The upper pattern *(+)←(read int x)* can still be grown while the pattern *(+)← (int lit)* has no more growth opportunities.

There is an additional pruning requirement to ensure that no pattern can be generated twice or incorrectly. Any location that can't be extended must be removed. From Figure 4.16 the third tree T3 is one such case, as both branches of the if statement have an assignment as child node. As node 37 becomes a node in the pattern, all locations containing it are removed. This prevents that the pattern has node 37 at two separate positions after the next growth phase. Figure 4.18 shows this process. After every growth, the growth map needs prune all nodes that exist to the left of the node that was injected (i.e., rightmost growth approach). This ensures that only ordered patterns are mined.

The previous examples only discuss mining of induced, ordered patterns. To mine embedded ordered patterns, i.e., the (⋆) wildcard, the algorithm works exactly the same, only changing the growth opportunities. For embedded mining these also receive all indirect locations of a node. The growth map for a single node represents a depth-first structure of its subtree. Whenever a node is grown, the growth map of the new node is removed from the parent node, in addition to itself and all growth opportunities to the left of it. To easily identify if the nodes are a direct or indirect relationship, all growth opportunities representing an indirect relationship are negative (e.g. -39 instead of 39). Because of the depth-first structure, this pruning process of the growth opportunities guarantees that the embedded patterns with a (⋆) wildcard are always distinct from induced patterns and do not share locations.

4.7 Mining in Compilers and Interpreters

Mining discriminative patterns according to any given metric still comes with two flaws that impact the significance of patterns that must be addressed:

Redundancy any discriminative pattern is likely to have siblings (patterns that are super- or subsets of each other) [35, 37]. The reason is the subgraph-supergraph relationship. I.e. an already discriminative pattern will always remain discriminative when grown. This leads to many redundant patterns identifying the same core information.

Frequent ≠ Significant Many patterns, especially smaller ones, occur very often in any search space but do not impact the functional or non-functional property mined. For example, most functions in code will have literals, but these literals are rarely relevant. To deal with this issue, research sometimes considers a *maximum support thresholds* in addition to a *minimum* threshold [35, 86].

The issue of redundancy is not solved easily. Any mining approach will produce redundant patterns, and the approach presented here produces even more redundancies through the generalization of nodes via a taxonomy. This is one of the reasons why Cheng et al. [37] suggest that smaller patterns are more useful, as the amount of redundancies grows with the size of a core pattern, as larger sizes mean a larger amount of possible relationships for a pattern to grow. The ways to solve these issues in this work is primarily filtering. This has been discussed in the *compact/closed* definition (see Section 4.6), which can reduce redundancies somewhat. Similarly, the (⋆) wildcard also ties smaller patterns together. Additionally, filtering of results can help. As these approaches primarily depend on how results are filtered and visualized, they are discussed in the second section of this work (see Chapter 10):

Filtering is already done via the applied metrics. However, filtering can also be done via overlap, e.g. when several patterns are similar enough only one is shown.

Merging as suggested by [40]. Patterns with enough overlap can be merged, although their score according to the metric should be similar.

Relationships Patterns have a contains and generalizes/specializes relationship. Using these relationships for visualization can help reduce information overload.

The following sections detail efforts to move from patterns mined as statistically significant towards patterns that are significant to the intent of improving functional or NFPs, all of them primarily tackle improving the likelihood that a frequent pattern is also significant. For this reason, the concept of *pattern verification* is introduced. An issue with granularity and frequency of patterns is solved via *co-located pattern mining* to find more complex sub-ASTs that are significant. Finally, *Outliers*, i.e., patterns that may decrease the significance of metrics or negate a general pattern, can be dealt with as well.

Pattern Verification

Pattern verification is not a new concept in the domain of pattern mining. Any pattern that has been mined is only worthwhile if it has been analyzed, and is shown to be responsible for the functional or non-functional impact that the pattern was mined for. However, in the context of working at the compiler or interpreter level, the verification of patterns can be automated. This automation process ties in with how ASTs are represented for mining. As this representation is taken directly from the system that has run the source code, and the execution outputs (test results) as well as observations are preserved, all information required to verify a pattern is available.

The automation stems from the use of GI. Anti-patterns, i.e. patterns that are identified to be negative, can be verified via the search space by selecting ASTs that contain the pattern. GI then can be utilized to maintain the rest of the AST, but remove the offending pattern and replace it with something different, possibly a matching pattern that was identified to fix the negative issue. Similarly, positive patterns, for example a pattern that fixes a bug, can be utilized in a search space containing a bug. The pattern can be applied via GI and checked if it really does fix the bug as expected.

Details of how this works on an algorithmic level are discussed in Section 5.3. As an example, consider the pattern that has been identified to fix the uninitialized variable access, the pattern *(write int)* was identified as discriminative in the positive search space (ASTs that have no run-time exception), and always occurs together with the pattern *(read int)* thus merged with the (⋆) wildcard. As the pattern was identified in the positive space, it is now applied to the negative space with GI. Running the modified ASTs shows that the bug has been fixed successfully in one of two AST, increasing the confidence in the pattern, as shown in Figure 4.19. The pattern is not completely valid, as in the second tree the pattern injects into the then path of the if statement, which is not executed before then else path containing the bug location.

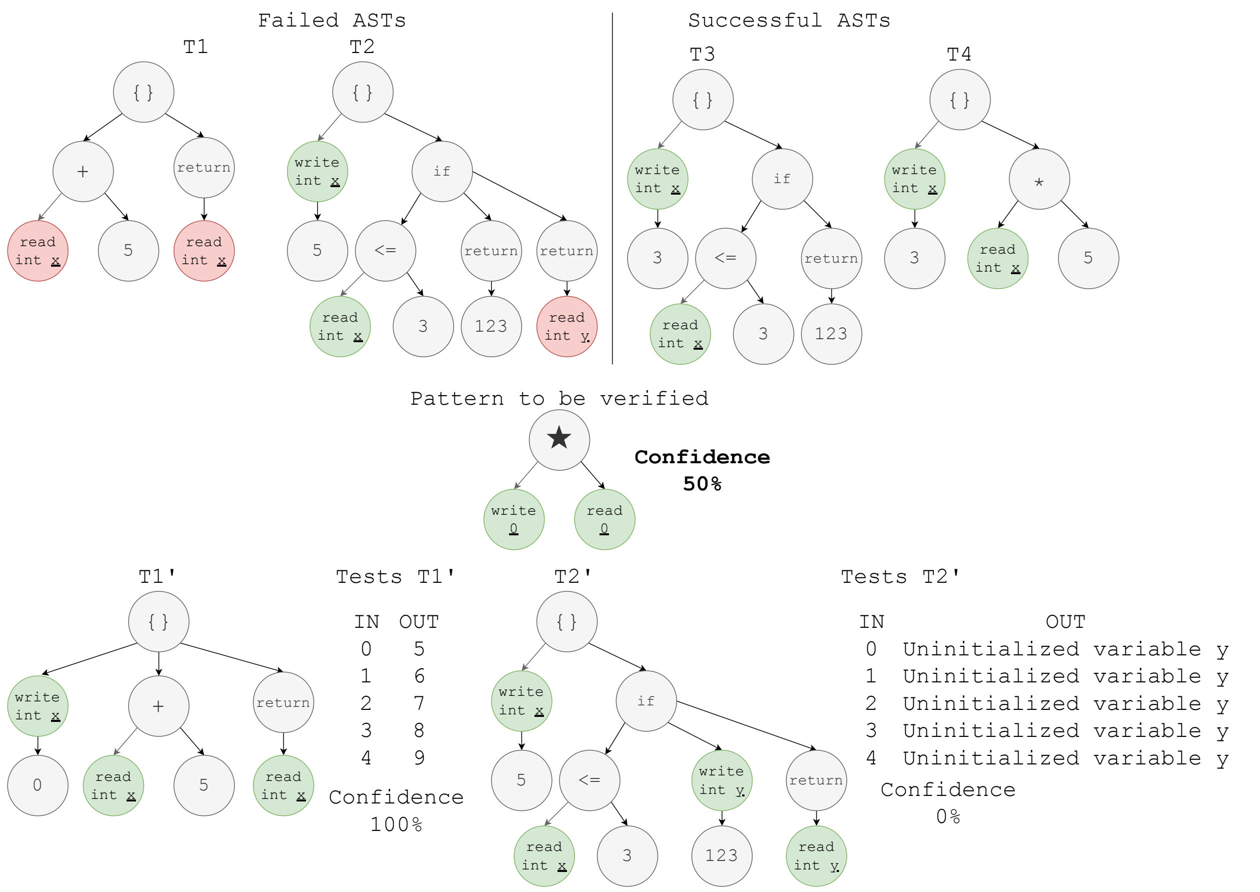
The example also shows that the confidence in a pattern is no guarantee that it is correct, as in theory the GI approach could have injected the *write* before the if statement resulting in a 100% confidence, which would not have been justified. To increase the quality of the confidence score, several mutations of each original AST need to be produced and tested.

Automated pattern verification can help via introducing a confidence score to a pattern (see Definition 4.7.1). The confidence itself varies greatly from the question the mining is supposed to answer, but generally is only applicable to Cluster Pattern Mining, and not significant subgraph mining. The above example considers the two clusters "failing" and "succeeding" ASTs. As the pattern was identified in the succeeding space, it is supposed to be applied to "failing" ASTs which after the modification should fit the criteria of the succeeding cluster, e.g., not failing the test cases anymore. The confidence score is calculated as:

$$\text{conficdence}(\text{pattern}) = \frac{\frac{\sum_{\text{test}} \text{test}[\text{successful}]}{\sum_{\text{test}} \text{test}}}{\sum_{\text{AST}} \text{AST}} \tag{4.6}$$

Definition 4.7.1 *The **confidence** in a pattern identified in a cluster X is how often it is applied successfully in an AST′, where the original AST lies outside the cluster X, and the modified AST′ containing the pattern lies in cluster X.*

In this way pattern verification helps to increase the likelihood that a pattern is significant and not just frequent. The concept can also help to remove redundant patterns. A pattern that has been identified to be significant throughout multiple experiments, but always has a low confidence can be identified and generally excluded. This enables, for example, the removal of smaller patterns, such as literals which usually represent outliers.



Tests T1'

```
IN  OUT
0   5
1   6
2   7
3   8
4   9
Confidence
100%
```

Tests T2'

```
IN           OUT
0  Uninitialized variable y
1  Uninitialized variable y
2  Uninitialized variable y
3  Uninitialized variable y
4  Uninitialized variable y
Confidence
0%
```

Figure 4.19: Within the search space at the top, the pattern *(write int)*← (⋆) →*(read int)* has been identified to fix the uninitialized variable access exception. The pattern is applied to the negative search space, with the modified ASTs) and execution results are shown at the bottom. As one AST still fails in one of two instances, the confidence is 50%.

Co-Located Pattern Mining

Literature considers pattern mining a single step process, assuming that patterns become evident through discriminative metrics in two different groups [36, 39, 87]. The base assertion is, that only patterns that occur in one group but not the other are of relevance. However, this is often not the case.

For example considering Figure 4.19, the pattern *(write int)*← (⋆) →*(read int)* would be found via discriminative pattern mining, as a successful pattern in the positive cluster. However, the actual reason the bug occurs is not identified, i.e. *(¬write int)*← (⋆) →*(read int)*. The *(read int)* itself occurs in both the positive and negative cluster and the *(write int)* is discriminative, as it occurs only in the cluster that has no failed executions. This shows one fundamental issue with discriminative pattern mining. When phrasing the example as a question, it would be *Why does the uninitialized variable exception occur?* The answer that discriminative pattern mining finds is *The uninitialized variable exception does* not *occur when a* (write int) *is present*, whereas the true answer should be *The uninitialized variable exception occurs when there is a* (read int) *operation without a preceding* (write int).

Pearson et al. [64] define this issue as a *fault of omission*, i.e. that a bug in a program is not caused by existing faulty code, but rather by the absence of code that should be there. 30% of changes to source code add new code to fix a bug rather than modify existing code. Discriminative pattern mining can identify such a fault of omission, but it can't identify a fix on its own.

A process to solve this in the mining process is cluster pattern mining. This process of mining attempts to find differences between the groups. This step answers the question *Is there a difference between several clusters that can explain a functional or non-functional property*? In many cases, this step is enough. If the identified patterns lie within the negative space, for example, an incorrect loop condition leading to endless loops, anti-patterns can be successfully identified that should be avoided to prevent the negative result. An example for the positive space is which list type should be chosen for run-time performance, depending on the ratio of read to write actions.

To achieve the answer of *What influences a functional or non-functional property?*, and *How can this be modified to change the functional or non-functional property?* a multi-step process is required leading to co-located pattern mining.

The Process of Co-located Pattern Mining

1. **Identify pattern size** In many cases ASTs in the same cluster are similar. Doing an initial mining per cluster with a *maximum support threshold* can help identify at which point the ASTs start to become different. This provides a good search area for mining discriminative patterns between the clusters.
2. Mine via **Cluster pattern mining** with the selected metrics and pattern target size to find discriminative patterns.
3. **Co-located pattern mining**
   a) **Identify co-located patterns in cluster** For each pattern deemed relevant, mine only the ASTs from its original cluster containing the pattern via significant pattern mining (one cluster), and find patterns that are located in the same tree. This step can be merged with b) when a metric is used that searches for a high support per cluster and a low discriminative value at the same time. These patterns can be combined via the (⋆) or (●) wildcards to create a target pattern.
   b) For each pattern deemed relevant **identify co-located patterns outside of cluster** and mine only the ASTs from its original cluster, and all other clusters. The metric should *prefer less discriminative patterns*, i.e. patterns that have a large overlap over the clusters.
   c) **Identify pattern replacements** likely pattern locations are those from outside the cluster mined in b). This means that if such a pattern is identified, it should be likely replaced or expanded to become the target pattern identified in a).
4. **Pattern Verification** can be conducted from this step on. From 2) anti-patterns can be removed, or patterns can be injected via GI. Alternatively, pattern replacements can be conducted using the patterns identified from 3)

It should also be mentioned that the process as described can lead to patterns that are improved over a singular discriminative pattern mining step. As an example, Step 2 of the process could identify that nested for-loops within ASTs often occur in trees that have a long run-time performance. As the secondary mining attempts to identify co-located patterns, the pattern may become larger by including the looping conditions, showing that the reason is an $O(n^2)$ behavior in the loop condition. Similarly, the process is able to identify outliers that would otherwise negate the pattern. As the pattern is now extended to include the conditions, a pattern containing nested for-loops in the search space for shorter run-time performance may be negated as its conditions are different.

Pattern Mining combined with Genetic Improvement 5

The application of genetic improvement in compilers and interpreters has already been discussed (see Chapter 3). The chapter had a focus on how to manage search spaces in that area, and how to deal with unviable solution candidates. All of this serves to produce Abstract Syntax Trees (ASTs) to let pattern mining (discussed in Chapter 4) identify recurring patterns for functional or Non-Functional Properties (NFPs).

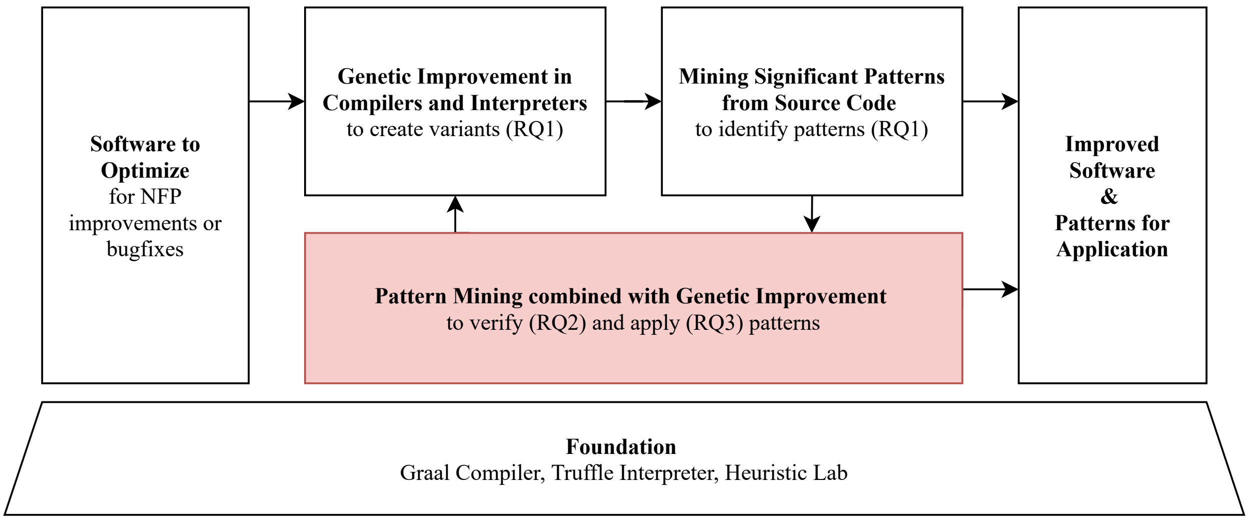
This chapter discusses the relationship between the two areas, and explains how Genetic Improvement (GI) can be utilized to mine patterns for a given type of functional feature, or NFP. This is primarily done via applying the fitness function towards these goals, as well as by modifying the search space via patterns.

The relationship built between the two fields can also be utilized to improve the confidence in a found pattern by verifying or disproving it using GI. This increases the confidence in found patterns[RQ2]. This can be done by intentionally attempting to produce counter-examples to disprove the pattern, or alternatively by attempting to introduce the pattern into ASTs and observing changes in behavior.

RQ2: How can the confidence in patterns be improved?

Found patterns can be applied directly in the GI algorithms to improve the quality of produced individuals. This is done via Knowledge-guided Genetic Improvement (KGGI), by introducing structural patterns into the syntax graph (see Section 3.4). This helps to prevent infeasible individuals in the population, i.e. anti-patterns. Alternatively, individuals that will contain patterns with a positive impact on a given NFP can be generated [RQ3].

RQ3: How can these patterns be utilized to lead to general optimizations?


Software to Optimize
for NFP improvements or bugfixes
Genetic Improvement in Compilers and Interpreters
to create variants (RQ1)
Mining Significant Patterns from Source Code
to identify patterns (RQ1)
Pattern Mining combined with Genetic Improvement
to verify (RQ2) and apply (RQ3) patterns
Improved Software & Patterns for Application
Foundation
Graal Compiler, Truffle Interpreter, Heuristic Lab

5.1 Types of Patterns

A pattern needs to be put into a context to determine how it is applicable in a search space, and how it impacts NFPs or functional properties. For example, a *pattern* that has a positive impact on run-time performance may be considered an *anti-pattern* for code-size or memory use. There are differences in how patterns can be applied. A pattern, in general, is something understandable to a human. But to make a pattern useable to change an AST, it must satisfy some conditions to make it applicable for automated processing.

Patterns always have to be considered with one specific NFP or functional property / bug in mind, to be considered a *pattern* or *anti-pattern* (Definition 5.1.1). In a different context, e.g. another NFP, the pattern may become an anti-pattern and vice versa.

Definition 5.1.1 *A* ***pattern*** *is an AST consisting of wildcards and nodes in a taxonomy that has a positive impact on one selected functional or non-functional property. If it has a negative impact, it is an* ***anti-pattern****.*

Both pattern types have their individual merits. Anti-patterns are often enough to improve the GI approach to limit the search space. The same goes for patterns, which can be introduced in GI to focus the search in more positive directions. Patterns may also be injected into a given AST without the need of an anti-pattern. For example, to introduce logging into a function, or to apply defensive programming.

Figure 5.1 shows an example for an anti-pattern that can be applied in KGGI. A *while* loop in which no variable that is *read* in the condition that also has a *write* in the body will lead to an endless loop. The pattern can be processed automatically since the branches of the pattern are defined (condition and body). If that were not the case, the pattern would be ambiguous to where the *read* and *write* should appear, as both appearing in the body would also be inferable without the clarified paths. Additionally, the pattern contains the anywhere wildcard (⋆) in both branches to allow the variable to occur anywhere under the condition or body. The underlined $\underline{0}$ is a reference to a variable. The name and type of the variable are irrelevant, but both the *read* and *write* node must access the same variable.

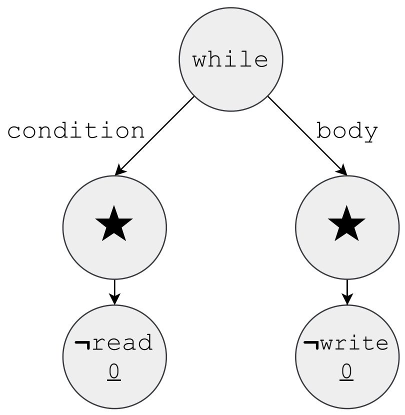


Figure 5.1: Example for an anti-pattern identifying endless loops. An endless loop occurs if there is no *read* in the condition that has a corresponding *write* to the same variable $\underline{0}$ in the loop body.

The meaning of the (¬) wildcard also needs to be clarified, as this is different for an anti-pattern or a pattern. In the case of an anti-pattern, the (¬) means that the absence of a node is an issue, e.g. in KGGI the node must be generated at this point. If writing Figure 5.1 as a pattern, both nodes would not have the (¬) wildcard, instead indicating that they are needed to be a positive influence on an AST.

The example also shows a possible future research direction. The wildcards in the approach can be more refined, to indicate more clearly where in an AST the best possible location for the fix would be, or that multiple options of sub-AST would be valid within one branch of the pattern. Considering Figure 5.1 the *write* in the loop condition may not be necessary if a break statement were in the method body. Furthermore, it is possible that patterns should be mined in different views of the source code. The pattern fails to account for branches, i.e., the control flow of the AST, making it possible that the pattern is satisfied with the *write* node being in an unreachable branch of an if statement. This either makes a wildcard necessary that is control-flow-sensitive or requires a pattern being identified in the control flow graph, also opening the question if the data flow graph and other representations should be considered in the future.

Another area to be explored in the future could be *rewrite patterns*, e.g., the combination of anti-patterns that are identified in a given AST and replaced with a positive pattern. This would improve the applicability of patterns in compilers and interpreters, as the patterns would be directly applicable to source code without the need for GI. Such rewrites have been suggested before in literature primarily to deal with refactorings based on a graph-based representation of source code [88–90].

5.2 Utilizing GI to Mine Patterns

GI can be utilized in two ways to mine patterns. The first is mining from experiments that were conducted successfully. This is especially promising in areas where the search-space is manageable and solutions can be found, which is often true for attempting to fix bugs in source code (see Section 6.3).

The second option of mining is to intentionally create experiments with an inverted fitness function. This means that instead of attempting to optimize a given AST towards run-time performance while preserving the semantic validity, the fitness function should instead attempt to increase the run-time performance while preserving the semantic validity. Similarly, the fitness function can be utilized to introduce bugs.

Attempting to decrease the functional features via GI can work well, given that literature already identifies that around 80% of solutions in GI are not compilable or produce a run-time exception [8, 9, 26]. What can also be considered, is to attempt breaking edge cases, i.e. modifying the function in such a way that only some parts of a given test suite fail and not all of them. This could lead to identifying mistakes in defensive programming or to find security bugs and identifying patterns with automatic fixes for them.

It is less obvious why attempting to decrease the quality of non-functional properties is easier than improving them. The reason for this is the mutational robustness of software [12]. Modifying source code via swap operations, code motion, or adding code has an impact on non-functional properties, with a lower likelihood of modifying the functionality of it. Schulte et al. [12] in this case report slightly more than 30% of all mutants being neutral concerning the functional properties of source code given a test suite, even including a delete mutation, which arguably has more impact than moving or adding code.

A serious limitation to the approach is, that artificial source code is significantly different from manually written source code [64]. This means that the patterns found can likely be applied to improve the GI approach, as this is a synthetic method to create new ASTs. Patterns may also be applicable to compilers and interpreters, as they need to be able to manipulate generic code in a synthetic way. Another limitation to this approach is that the NFP modified via a fitness function for them will concentrate in one area, either the negative or positive. This is an issue when discriminative pattern mining is applied, as this approach needs positive and negative samples to work. If the experiment is done purely for mining, another option is to design fitness functions only in the space that should not be mined in order to gather a wider range in the NFPs or functional properties for mining.

An alternative for utilizing GI to mine patterns that are useful for developers would be mining software repositories [91–94]. This would provide the advantage of mining patterns concerning faults and NFP as introduced by developers, but provides a disadvantage in having to make the code executable and evaluating functions with minimizing side effects. In addition, this would prevent finding changes that can be done to source code at the level of an interpreter or compiler, which may not be an option in a pure text representation, or not applicable to the compiler or hardware setting the repository is usually applied in.

5.3 Verifying or Disproving Patterns via GI

The verification of a pattern is done to improve the confidence that the pattern works as expected, or to identify in which context a pattern is applicable. Source code is the sum of its parts, so often one pattern is not solely responsible for one exception or for the run-time behavior of an AST. Instead, there may be multiple patterns leading to the same outcome, or a group of patterns responsible for an NFP.

In general, this makes anti-patterns more easily provable than patterns with a positive influence. Identifying the anti-pattern leading to uninitialized variable access would be that a *read* occurs without a given *write*. All ASTs matching this pattern will produce the exception, making the pattern easily verifiable. Similarly, a pattern with a negative influence on run-time performance, for example nested for-loops iterating over an entire collection in both loops, can more easily proven to be a negative influence via its existence. A positive pattern that is supposed to improve the run-time performance or to repair a bug is not as easily provable. Chapter 6, for example, shows bugs identified with a confidence of above

90% but corresponding fixes often had different bugs occurring instead of the fixed bug.

The verification can be approached in one of two ways. Modifying existing ASTs or via GI. Both approaches have their advantages and disadvantages.

Modifying an existing AST is the preferred way to prove a pattern, as this allows a more direct comparison of the influence of the pattern on its own, i.e., if only the pattern is introduced, it should be solely responsible for the changes in functional and non-functional behavior of the modified AST. Attempting to apply the same pattern in multiple ways to a single AST is also a good way to increase confidence in the pattern or to identify its limitations. A pattern can be applied at multiple locations, especially if it contains wildcards or more generalized nodes from a taxonomy.

For anti-patterns the preferred approach is GI. The reason for this is that there is more confidence in the pattern itself, as not only known solutions are explored, but many AST combinations and mutations are being explored. This still introduces some risks, as not every AST may have the intended effect. For example, when attempting to research the nested for loop pattern, many ASTs may have bugs introduced that make them fail instead of just increasing the run-time performance. The important consideration is that all AST containing the anti-pattern should fail or have the expected negative effect. If a bug-pattern is being explored, this means that no AST in the population should succeed, but not necessarily all failures need to be accountable for the specific bug. Concerning the nested for loops, this means that either the individuals should fail or be semantically valid and have an increased run-time compared to other valid ASTs not containing the explored anti-pattern.

This in turn means that verifying patterns via GI should be done via two experiments and not one. Anti-patterns are verified via excluding them explicitly in one GI experiment, to create many valid mutants not containing the pattern, to be compared with mutants containing the pattern from a different experiment where GI was specifically tuned to contain the pattern. How patterns can be introduced into the syntax graph is discussed in Section 5.5. For the purpose of pattern verification, the only thing that needs to be done is modifying the *mutation operation* of the genetic algorithm. The mutator should either generate the pattern, or at least have a high probability of it being introduced. The crossover operation can be modified to ensure that the pattern is not removed via crossover. Attempting to introduce a pattern via crossover might not be possible if the required nodes are in none of the ASTs selected to be crossed.

Of course, this approach of verifying anti-patterns has the issue that other patterns of code can lead to a similar behavior in NFP or to the same bug. This is the reason why it is primarily suitable for anti-patterns which are less difficult to verify in the negative space. The experiment where the anti-pattern was excluded may show other anti-patterns with similar behavior that could falsely indicate that the pattern is not responsible. This can be somewhat mitigated by applying co-located pattern mining in the result of both experiments and finding and excluding additionally identified anti-patterns in follow-up experiments.

The approach via KGGI is not suited to mining positive patterns, as those are more likely to be affected by other patterns. This can lead to side effects when trying to discern if the pattern is responsible for an improvement. The pattern may not impact the feature under test because other parts of the source code counteract the positive effect of the pattern. What such a dual-experiment set up can help with, is finding co-located patterns or anti-patterns that may also influence the same function or NFP.

Another challenge is the general applicability of identified patterns. For example, a pattern may have just been identified because the entire population of KGGI was stuck in a local optimum. To mitigate this issue, GI can be used with a *diversity* fitness function, i.e., a function that contains a measure of how different ASTs are from each other in the current population, or over all populations. Such a diversity measure has been previously used in Genetic Programming (GP) to prevent premature convergence of the algorithm and for maintaining diversity to improve the evolutionary approach [95–97]. In this case, it could be used to drive the trust in the analysis of a pattern to diversity of the AST generated in testing. As this work utilizes multiple mutations instead of GI to verify patterns, diversity measures are not applied, but present an opportunity for future research when analyzing the effects of multiple patterns in GI experiments.

5.4 Context of Patterns

Patterns are often valid only in a specific context. This has been previously discussed in the context of taxonomies (see Section 4.2) that are built for a specific purpose such as mining patterns concerning data types or patterns concerning access to variables. Similarly, patterns are identified with a specific purpose in mind, such as fixing bugs or adding defensive programming measures. Similarly, patterns may improve NFP, and frequently there is a trade-off between them. An important aspect of applying the presented algorithms and concepts lies in the quality of the data. This requires recording the NFP measurements during tests reliably, similar to how recording the ASTs and tests for them is critical. Measures that have been taken to achieve an acceptable quality for experiments in this work are discussed in Chapter 10.

This is especially important for mining patterns, considering what ASTs will be used to search. Due to the approach being based on *mining significant patterns*, in the form of often recurring sub-ASTs, the likelihood of useful patterns is increased with a search space that is used for the same purpose, and thus has to be selected manually. For example, one experiment can consider sorting algorithms with a taxonomy for data types to identify interesting patterns in that area. Mixing these sorting algorithms with math functions (square root, etc.) that may be optimized via completely different patterns using approximate computing would likely have a negative impact on the quality of the found patterns.

A hitherto not specifically addressed consideration can be the signature of a function, specifically its input and output. While this is beyond the scope of our work, function signatures may imply a similar use context and similar patterns. The use of collections as input to a function is a likely indicator that these collections will have to be fully or partially processed, having a higher probability that patterns with looping structures will occur. This correlation may in the future be used as an indicator if different ASTs can be used as a search space to identify new patterns.

5.5 Improving Genetic Improvement With Mined Patterns

KGGI, specifically the syntax graph of it, can be improved via mined patterns, and anti-patterns (see Section 3.4). This section describes the necessary algorithms to enrich a syntax graph with anti-patterns and patterns, based on injecting requirements for the creation or prevention of specific nodes. For a mutation, i.e. for creating a new AST, this works by transferring this information from a pattern into the syntax graph. To support selection and crossover, a selected AST is transformed into a list of requirements that they have to still be considered valid in a new context.

Restricting Anti-patterns

Anti-patterns are patterns that should never occur in a tree. Some of these anti-patterns should always be injected into the syntax graph to prevent completely infeasible individuals, i.e., those that will lead to run-time exceptions, for example, uninitialized variable reads. Others depend on the execution context, and should be added in that context, for example patterns restricting multiple loop encapsulations when optimization towards run-time performance is attempted.

Algorithm 9 works because the syntax graph of KGGI follows a left to right, bottom up creation approach. This means that strategies creating nodes closer to the root or strategies creating nodes further to the left can impose restrictions on anything that is created later, i.e., further down, or to the right. This is achieved by injecting *do not create requirements* at the root of an anti-pattern. Note that the algorithm is highly abstracted and actually happens over different strategies in the syntax graph (see Figure 3.9).

During the canCreate phase of the syntax graph of any called strategy, the strategy must consider all applicable anti-patterns (loop starting with Line 1).

($\star$) wildcards allow skipping nodes, so their currently active node must be validated (Lines 5-9). In case the pattern actually matches, the next node in the pattern must be set active (Lines 21 + 22).

($\neg$) negation wildcard nodes in a pattern do not lead to restrictions in a pattern, instead they indroduce *create requirement*s. If a strategy is presented with such a requirement, it checks if it can fulfill the required node or if later strategies may be able to create the node instead (Lines 10-14). Checking if later strategies allow anti-pattern prevention is done to ensure the search space is not restricted too much. If every anti-pattern or pattern were to be satisfied at the first opportunity, the search space would be severely restricted.

Algorithm 9: Update child node requirements to prevent anti-patterns.

```
Data: antiPatterns
Data: context
Data: nodeType
/* Process all matching anti-patterns and root wildcard
   patterns.                                                    */
foreach $ap \in antiPatterns$ do
    match ← ap.node;
    anyWildcard ← False;
    created ← False;
    if $ap.node = \star$ then
        /* Select active child of ★.                             */
        match ← ap.node.children.active;
        /* Check if current node is relevant for node type.
           */
        if $match.type \neq$ " nodeType then
            continue;
        anyWildcard ← True;
    if match.startsWith(¬) then
        /* Add create requirement if necessary.                  */
        if ¬validateDegree of Freedom (DOF)() then
            req ← createRequirement(ap);
            context.requirements.add(req);
            created ← True;
    else
        /* Add do not create requirement if necessary.           */
        if ¬validateDOF() then
            req ← doNOTcreateRequirement(ap);
            context.requirements.add(req);
            created ← True;
    end
    if $anyWildcard \wedge created$ then
        /* Ensure that ★ wildcard is updated.                    */
        updateActiveChild(ap)
end
Result: context
```

For regular nodes the strategy can either check if the anti-pattern in question has later nodes that should not be created, in which case it may opt to ignore the requirement, and instead replace it with a *do not create requirement* of a later node (child or sibling). This allows us to partially create anti-patterns which are never introduced completely (Lines 15-20). This is also done to ensure that the search space is not overly restricted.

The restriction process has one caveat, which stems from its advantage of pruning infeasible search space parts. Not all possible permutations of a given anti-pattern are necessarily negative, if they aren't detailed enough. For example, preventing that variables don't have a connected write (anti-pattern $(\neg write) \rightarrow (read)$), is not always correct. It is only correct in the context of local variables. Global variables may have been correctly initialized outside the currently modified AST.

Additionally, the restriction has an impact on the search space, and using several anti-patterns at the same time can unintentionally prevent needed parts of the search space due to the overlapping exclusions of anti-patterns. This mostly impacts node types that will be prevented from creation. It also can impact anti-patterns if one anti-pattern requires a (¬) node to be created, while another anti-pattern has the same node without a (¬), making only one of the two anti-patterns fulfillable. In such a case, KGGI attempts to satisfy as many patterns as possible instead of all patterns.

Even though anti-patterns can restrict a search space beyond the intent, the approach provides an advantage of preventing infeasible individuals in the population and raising the average quality of the solutions. As a consequence, this improves the chance that new AST modifications are found that show different NFP than previous ones. This in turn enables mining to find new patterns.

Injecting Positive Patterns

Positive patterns, known to occur often in ASTs that are good in a functional or NFP, are injected into the search space. This works by introducing requirements into the syntax graph. Similar to anti-patterns, positive patterns can prevent each other from occurring, i.e., one pattern is possible but not the other. This is because some requirements may directly contradict each other. For example, *requirement A* states create write 0, and *requirement B* states do not create write 0. *Creation requirement*s are given to strategies being called during the creation process. For patterns, this happens more often than for anti-patterns, as any node that has no (¬) requirement will be created via *creation requirement*.

Algorithm 10, shows how a pattern is processed in the syntax graph, inducing new *creation requirement*s to ensure child nodes of the pattern will be generated. The algorithm essentially injects for the root of a pattern (if it is not a wildcard node) a check in the strategy corresponding to that node. If the pattern node equals a specialized class, this is a copy of the original strategy without any pattern restrictions. If the strategy handles a generalized node from the taxonomy, it will instead inject into the strategies of all nodes that match this taxonomy part. The (¬) wildcard is not a valid root node for positive patterns. The (⋆) however is a valid root and is injected into every single strategy. This happens before the algorithm (not shown).

Every pattern has a random activation chance between 0 (never activated) and 1 (always activated). To ensure that strategies that compete with each other (e.g., have the same root node) do not prevent each other in child-nodes only one strategy can be activated at the same time. For example, if three strategies exist with an activation chance of 1, only one of them will be chosen at random (Lines 3 and 4). The (⋆) wildcard is handled exactly in the same way as when dealing with anti-patterns (Lines 7-11 and 23-24). If the pattern matches, the active child of the wildcard will be updated, otherwise the requirement remains unchanged to be satisfied by a later node.

Each of the strategies injected with a pattern root *creation check* creates new *creation requirement*s for the child. This requirement is essentially -

pattern x - node n, where x is the pattern, and n is the next node that the pattern shall match. This requires the syntax graph to continue editing the pattern on every creation with a syntax graph strategy (Lines 12-16). This does not prevent multiple patterns from being active at once, as the strategies needed are independent of each other. For example, a child node that is a valid root of another pattern is allowed to introduce an additional pattern; only patterns with the same root node prevent each other. To prevent endless recursion, one specific pattern can only be active once and only be active again after it has been completely created (not shown).

Algorithm 10: Update child node requirements to inject positive patterns.

```
Data: patterns
Data: context
Data: nodeType
/* Process all matching patterns, and ★ root patterns.   */
1  created ← False;
2  foreach ap ∈ patterns do
       /* Ensure at most one pattern randomly activates.   */
3      if created ∨ random.next() > ap.activationChance then
4          continue;
5      match ← ap.node;
6      anyWildcard ← False;
7      if ap.node = ★ then
           /* Select active child of ★   */
8          match ← ap.node.children.active;
           /* Check if current node is relevant for node type.
              */
9          if match.type ≠ ″ nodeType then
10             continue;
11         anyWildcard ← True;
12     if ¬match.startsWith(¬) then
           /* Add create requirement if necessary.   */
13         if ¬validateDOF() then
14             req ← createRequirement(ap);
15             context.requirements.add(req);
16             created ← True;
17     else
           /* Add do not create requirement if necessary.   */
18         if ¬validateDOF() then
19             req ← doNOTcreateRequirement(ap);
20             context.requirements.add(req);
21             created ← True;
22     end
23     if anyWildcard ∧ created then
           /* Ensure that ★ wildcard is updated   */
24         updateActiveChild(ap)
25 end
Result: context
```

The creation check has two separate options. The first option is that it only triggers if the matching create requirement exists, essentially activating or deactivating paths that are currently needed for the pattern to continue. Alternatively, the check can validate if the previous nodes in the pattern already exist, only then the next pattern node is injected.

This only happens if a sub-AST is being created where an existing AST already contains part of a pattern. This is handled by injecting the context of the existing AST via Algorithm 11. It is mostly needed during mutation operations if a pattern must remain in the AST, but is partially mutated, and crossovers if the pattern must remain intact.

Similar to the restriction of anti-patterns, patterns may also contain (¬) wildcards. These are handled by checking in the subsequent adjacent nodes that they were not created. Previous nodes inject a *do not create requirement*, but will not remove the relationship outright to leave some leeway in the search space (Lines 18-21). If the optional requirement was not injected, and the (¬) node was injected, a following strategy that is part of the pattern will remove all requirements related to the pattern from the context, since the pattern cannot be continued. The injection of the *do not create requirement* can be changed to be required if it is a pattern that must be created (activation chance 1).

Injecting pattern nodes into the syntax graph influences the direction into which the search space evolves. However, it does not outright restrict the search space as long as all patterns remain optional (activation chance <1). While positive patterns do not decrease the amount of infeasible individuals, they can help to improve the overall quality of the population and to focus the search in areas known to do well for a given problem type.

Reducing a Subtree to Requirements

For the creation of a new subtree via the syntax graph, the surrounding AST into which the new sub-AST is injected must be considered. This also applies to completely new ASTs in relation to the entire program (heap, other functions). The goal is to ensure that any AST being created is valid. Similarly, crossing two existing trees must ensure that the crossover does not break the source code in a way that would prevent execution. This means that anti-patterns need to be considered during crossover.

Considering the context can be achieved by reducing a given partial AST to a set of requirements that it needs to uphold. This is always seen in the context of the *remaining* part of the AST, e.g. the AST being mutated without the AST replaced by mutation. For crossover, this is the AST being crossed into, without the subtree that will be replaced. Vice versa, the sub-AST selected to cross into this place must also be analyzed for anti-patterns.

Algorithm 11 performs such a reduction, essentially via a simulated creation approach. It is a part of the syntax graph and iterates through the AST asking the root strategy to return unfulfilled requirements. This is done essentially via the same functionality that the previous algorithms for anti-pattern and pattern injection outline. Instead of choosing a strategy to create the selected nodes, the syntax graph root strategy selects the strategy that was responsible for creating it. This is usually unique, as all executable node classes usually have only one dedicated strategy.

A strategy selected by the root strategy adds its requirements to child and later nodes in the AST. Due to the depth-first iteration of the algorithm, later strategies for nodes will resolve the same requirements they would remove when creating the node.

Algorithm 11: Recursively reduce AST to a set of unfulfilled requirements.

```
Data: node
Data: requirements
Data: patterns
Data: antiPatterns
1  context ← requirements.get(node.parent);
2  currentContext ← Context.create() patterns.addAll(context.patterns);
3  antiPatterns.addAll(context.antiPatterns);
4  foreach ap ∈ patterns do
       /* Use the anti-pattern restriction to find anti-pattern
          requirements.                                          */
5      currentContext.add(Algorithm 10)
6  end
7  foreach ap ∈ antiPatterns do
       /* Use the pattern injection to find pattern
          requirements.                                          */
8      currentContext.add(Algorithm 9)
9  end
10 foreach node ∈ node.children do
       /* Recursively call this algorithm for all nodes.         */
11     Algorithm 11(node, requirements)
12 end
13 requirements.add(node, currentContext);
Result: requirements
```

After all nodes in the AST are processed, the root strategy selects a change point *change point* (either mutation or crossover point) which defines where a subtree will be modified. Requirements for all nodes under this point are removed. For example, an injected write to a variable is removed, so later nodes do not mistakenly identify the variable as initialized. All following nodes validate if their requirements are satisfied or not. This is done by the active strategy for the node checking its *creation check* mechanics. If requirements are not satisfied, they add an *unsatisfied requirement* to the context.

If Algorithm 11 fails for a node, this means that the strategy is not valid, in which case the requirements are unresolvable. This can happen in one of two cases. The first is that multiple strategies exist due to manual manipulation of the AST. The other is introduced via grafting, i.e., via ASTs that were introduced from other parts of the source code or from other experiments. As these grafts weren't necessarily created with the same syntax graph, they may not adhere to it and the requirements can't be extracted. The algorithm in this case skips the node and ignores its influence on requirements.

If multiple strategies can be selected, due to manual syntax graph manipulation, selection of one strategy can resolve the issue. The following selection mechanisms can be used:

Random A random selection leaves open the most chances for the search space.

Least Requirements Can be calculated in one of two ways. The fast evaluation is to evaluate all possibilities and select the one introducing the least amount of requirements. The full evaluation is to run the algorithm for the remainder of nodes through the AST and choose the option with the least amount of requirements. Usually, the fast evaluation is the better choice, as a full evaluation can lead to uncontrollable search spaces if multiple nodes have multiple options.

Pattern Tracing can be done only for ASTs that are created with the syntax graph. The strategies can add tracing information to the nodes they create, immediately allowing a match when they are analyzed again at a later point. This does not work for grafted nodes.

Pattern pre-identification Can be done for all patterns, and must be done if the pattern is required to be in the individuals for the GI experiment. Instead of letting the selection happen during the requirement reduction, Algorithm 11 is used on the full AST (before removing a subtree in mutation or creation), and one is selected. Nodes that are in parts of the AST that are to be removed will be added as requirements to the syntax graph before Algorithm 11 starts with the creation point.

Experiments | 6

The research questions of our work are answered via an empirical evaluation described in the following sections.

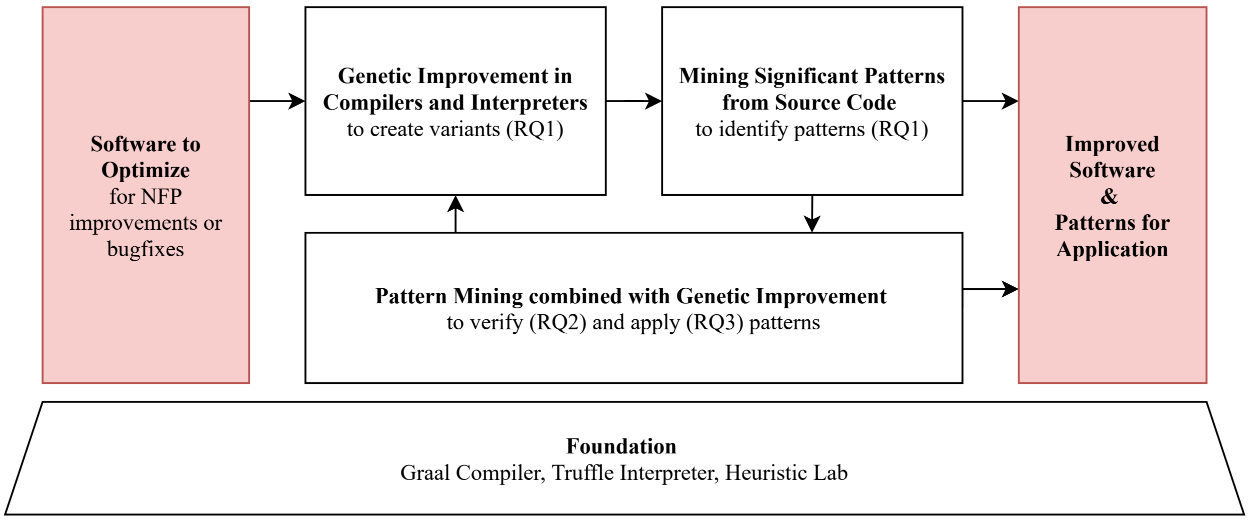
A total of 25 Abstract Syntax Trees (ASTs) have been chosen for exploration. All of these functions are executed in MiniC (see Chapter 9), to ensure comparability :

Math Algorithms A suite of math algorithms, all based on the concept of Newton Raphson approximation [98], was chosen, as these algorithms provide a set of relatively compact ASTs and require a high mathematical precision. These algorithms also have been a subject of study in Genetic Improvement (GI) [99–102]. Previous research also has been done by the author in collaboration with others, successfully generating lookup tables to improve the accuracy of these functions in comparison to the C, C++ and Java libraries, and outperforming C++ and Java concerning run-time performance [103–105]. This work however has not been conducted in MiniC.

Sorting Algorithms Sorting Algorithms contain larger ASTs compared to the math algorithms. The different well-known algorithms in this domain have a varying run-time complexity (and thus performance) to solve the same problem. Similar to math algorithms, previous work on these algorithms exists in Genetic Programming (GP) [12, 106].

Neural Network A neural network with several variations and activation functions has been chosen as the final set of ASTs. In addition to having large ASTs, they represent approximate computing, meaning that the solutions generated via GI do not need to exactly reproduce the functionality of their original counterparts.

The following sequence of experiments will be described in the next sections of this chapter.



Software to Optimize for NFP improvements or bugfixes
Genetic Improvement in Compilers and Interpreters to create variants (RQ1)
Mining Significant Patterns from Source Code to identify patterns (RQ1)
Pattern Mining combined with Genetic Improvement to verify (RQ2) and apply (RQ3) patterns
Improved Software & Patterns for Application
Foundation
Graal Compiler, Truffle Interpreter, Heuristic Lab

To begin with, an initial analysis of the test set is shown, in order to further explain the different scenarios under test and to provide an overview of the Non-Functional Property (NFP) of run-time performance in each of the 25 ASTs. The test suites, which are used to verify the semantic validity of ASTs in the GI experiments, as well as the benchmarks used to evaluate the run-time performance are explained in this section.

Following this, for each of the 25 ASTs an experiment is conducted as a baseline. This experiment showcases Knowledge-guided Genetic Improvement (KGGI), but without any identified patterns or anti-patterns. It serves as a base to *mine mutational bug patterns* that often occur in GI. This mining is the next experiment in line. It serves to partially answer the primary research question ^{RQ1}. Specifically, how *mutational bug patterns* influencing the functionality of GIs can be identified.

RQ1: How can recurring patterns be identified that impact or improve a functional or Non-Functional Property?

In the experiment, the patterns are also validated and assigned a confidence^{RQ2}. This is done to identify if anti-patterns are responsible for bugs, and if the corresponding prevention mechanisms work.

RQ2: How can the confidence in patterns be improved?

The experiments using KGGI are then repeated with the identified patterns and anti-patterns. This serves to further improve the confidence in the bug patterns by evaluating if the identified bugs occur more or less often. This also shows how the found patterns can be applied GI^{RQ3}.

RQ3: How can these patterns be utilized to lead to general optimizations?

Finally, the series of experiments is concluded with evaluating the performance of ASTs generated by KGGI that are semantically equivalent, as validated via a test suite, to their original solution. These performance evaluated ASTs are then mined for performance patterns, to answer the second part of RQ1, concerning patterns for NFPs, and to check if general optimization patterns can be identified.

6.1 Initial Analysis of the Experiment Set

All experiments were conducted on the same test set of 25 ASTs, grouped into three categories.

The *math* algorithms contain 8 different ASTs. Of these, 7 implement variations of the Newton-Raphson approximation method [98]. The final algorithm *Square root - Java* (2nd order root), one of three different ASTs versions to compute the square root, uses a MiniC Builtin node that calls the Java Math.sqrt() function. It primarily serves as a comparison between calling native functionality of the JVM vs. implementing it manually in the Truffle guest languages. For comparison, two different implementations of square root exist. The first *Square root - lookup table* is a reproduction of Krauss and Langdon [104], utilizing a lookup table for the initial guess used by Newton-Raphson. Their lookup table guarantees double precision (precision down to the last bit of a double value), with just three iterations. The reproduction guarantees float precision, as all algorithms either use integer or float as their base data type. *Square root - regular* in comparison requires 30 iterations to guarantee float precision over the entire range of the float values. *Cube root* (3rd order root) requires 50 iterations, and *super root* (4th order root) requires 60. The inverse square root ($1/sqrt(x)$) also uses 50 iterations. The final two algorithms, *Logarithm 10* and *Logarithm Natural* both do not use a fixed

Table 6.1: Overview of the **math functions** in the experiments. The node count equals all nodes available for optimization and mining. The invocations count how many functions are invoked during execution.

Function	**Node count**	**Invocations**	**Amount of Tests**	**Notes**
Square root - java	8	1	10	Call to Java implementation.
Square root - lookup table	42	3	16	Function, derivative and lookutable position called
Square root - regular	36	2	10	Function and derivative called
Cube root	39	0	10	
Super Root	43	0	10	
Inverse Square Root	38	0	10	
Logarithm 10	60	1	10	Loop checks remaining error
Logarithm Natural	56	1	10	Loop checks remaining error

iteration, but rather loop until the remaining error is less than 0.000001. In addition to the two different strategies to achieve precision (fixed loops, remaining error), the implementations contain a different amount of invocations. *cube root* and *super root* have all functionality inlined, whereas the other functions call some functionality such as checking the absolute value, calls to the lookup table or the function / derivative used by Newton-Raphson.

Except the *square root - lookup table* function, which works for a range between 0.5 and 2.0, all functions were tested on the same input. For all functions, the test set contains 10 tests with a corresponding floating point input and output.

A total of 10 *sort* algorithms comprise the second test set. It consists of 8 different sorting algorithms, and two duplicates with different implementations. Both *merge sort* and *selection sort* have two versions implemented. One version with function calls to parts of their functionality, and another with all calls inlined. This was done to compare the effect of partial functionality not being present in the AST under optimization. The AST sizes range between 56 and 206 nodes. As all algorithms fulfill the exact

Table 6.2: Overview of the **sort functions** in the experiments. The node count equals all nodes available for optimization and mining. The invocations count how many functions are invoked during execution.

Function	**Node count**	**Invocations**	**Amount of Tests**	**Notes**
Bubble sort	62	0	5	
Heap sort	165	0	5	
Insertion sort	56	0	5	
Merge sort	70	1	5	Call to merge
Merge sort inlined	152	0	5	
Quick sort	111	2	5	Call to partition and swap
Quick sort inlined	206	0	5	
Selection sort	56	0	5	
Shaker sort	118	0	5	
Shell sort	91	0	5	

Table 6.3: Overview of the **neural network functions** in the experiments. The node count equals all nodes available for optimization and mining. The invocations count how many functions are invoked during execution.

Function	Node count	Invocations	Amount of Tests	Notes
Rectified Linear Activation	587	2	1	
Leaky Rectified Linear Activation	599	2	1	
Sigmoid	587	2	1	
Swish	607	2	1	
Tanh	635	2	1	
Fully Inlined NN	657	0	1	Sigmoid activation function
NN with all Activation Functions	545	4	1	All activation functions available, but only one used

same goal, *sorting integer arrays*, this test set also serves to investigate the size of the AST that KGGI and Independent Growth of Ordered Relationships (IGOR) can be utilized on. All algorithms are tested on the same 5 randomly sorted int arrays to enable accurate comparison of their run-time performance.

The final test set comprises 7 different versions of a AST neural network. The neural network algorithm itself is the same in all versions. Only the core activation function is different for all versions. Five well-known activation functions *sigmoid*, *swish*, *tanh*, *rectified linear activation* and *leaky rectified linear activation* are selected. The remaining two AST versions are one inlined version using the *sigmoid* activation function, and one version that allows invocation of all activation functions, using *swish* in the implemented call. All versions are tested on the same in- and output set. One test attempts to create an XOR gate. This test was selected primarily to restrict the demand of the performance evaluation, and was intentionally chosen as it executes on all implementations with less than 10ms run-time. As the profiling of the run-time consists of 200,000 repetitions of the test, this means that each AST performance evaluation will take less than 34 minutes to run. This may not seem much for the original 8 ASTs, but the KGGI tests in the subsequent sections can produce up to 2,000 unique ASTs that need to be measured, and these ASTs can be less performant than the original implementation.

During the initial performance profiling, some odd behavior was ob-

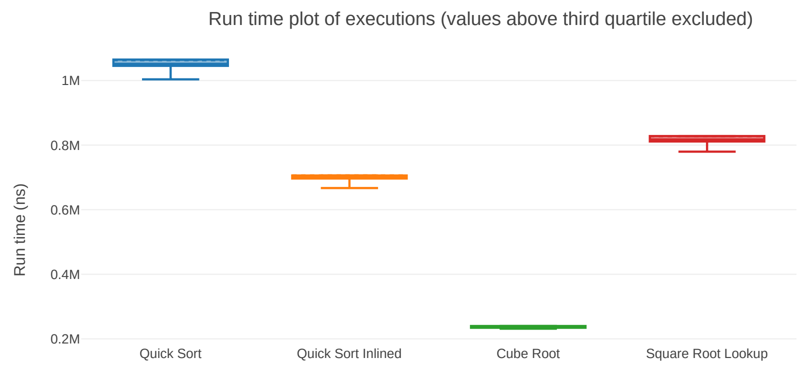


Figure 6.1: A previously undetected bug in MiniC caused issues with invocations. Quick Sort inlined is 30% faster than regular quick sort. Cube Root with 50 iterations is much faster than the square root lookup function with only 3 iterations.

served. Functions with invocations were much slower than those that did not contain invocations. Figure 6.1 shows four functions where this issue is glaringly obvious. Quick Sort with two invocations to the partition and swap functions takes 30% more run-time than the exactly same code with these functions manually inlined. Both versions of Quick sort are iterative, and the only difference is the manually inlined functions. The cube root function which takes 50 iterations for its calculation is much faster than the square root which has calls to a lookup table, the function and derivative.

The four algorithms were mined via a differential pattern mining using IGOR, grouped by:

imperformant quick sort, square root lookup table
performant quick sort inlined, cube root

The only difference that was identified, are the *invoke* nodes in the *imperformant* group. The Graal compiler does inlining as an optimization, which means that this anti-pattern should not be the cause of the performance differences. When utilizing the debugging options of the Graal compiler [1], the compilation trace indeed shows that an inlining is done for both *quick sort* and *square root - lookup table*.

1: polyglot.engine.TraceInlining=true

```java
@NodeInfo(shortName = "write-local-int", description = "Writes int to stack")
@NodeChild(value = "valueNode", type = MinicIntNode.class)
public abstract static class MinicIntWriteNode extends MinicWriteNode {
    @Specialization
    protected void writeInt(VirtualFrame frame, int value) {
        // The following line assigns the data type for a frame slot. With this line absent the performance is severely impacted.
        frame.getFrameDescriptor().setFrameSlotKind(getSlot(), FrameSlotKind.Int);
        frame.setInt(getSlot(), value);
    }
}
```

Listing 6.1: Problem causing the issues with inlining, and the fix to repair it.

Figure 6.2 shows two Graal IRs for square root after it has been optimized by the compiler already. In the left graph the inlining was done manually, in the right graph no inlining is conducted. Both versions do not contain any invocations to other functions anymore. The right side however contains many more *FrameWithoutBoxing*, also known as materialized frame. These types of frames are primarily used for the heap, where global variables are stored. Apparently, Graal is being prevented from optimizing the arguments given to the original invocation, forcing it to introduce materialized frames. The true reason behind this is the *GuardedUnsafeLoad* which can be seen multiple times in both sides of the figure, whenever a reading access to the variable happens. This unsafe load needs to happen when the compiler does not know what data type is stored in a given frame slot. Listing 6.1 shows the applied repair for

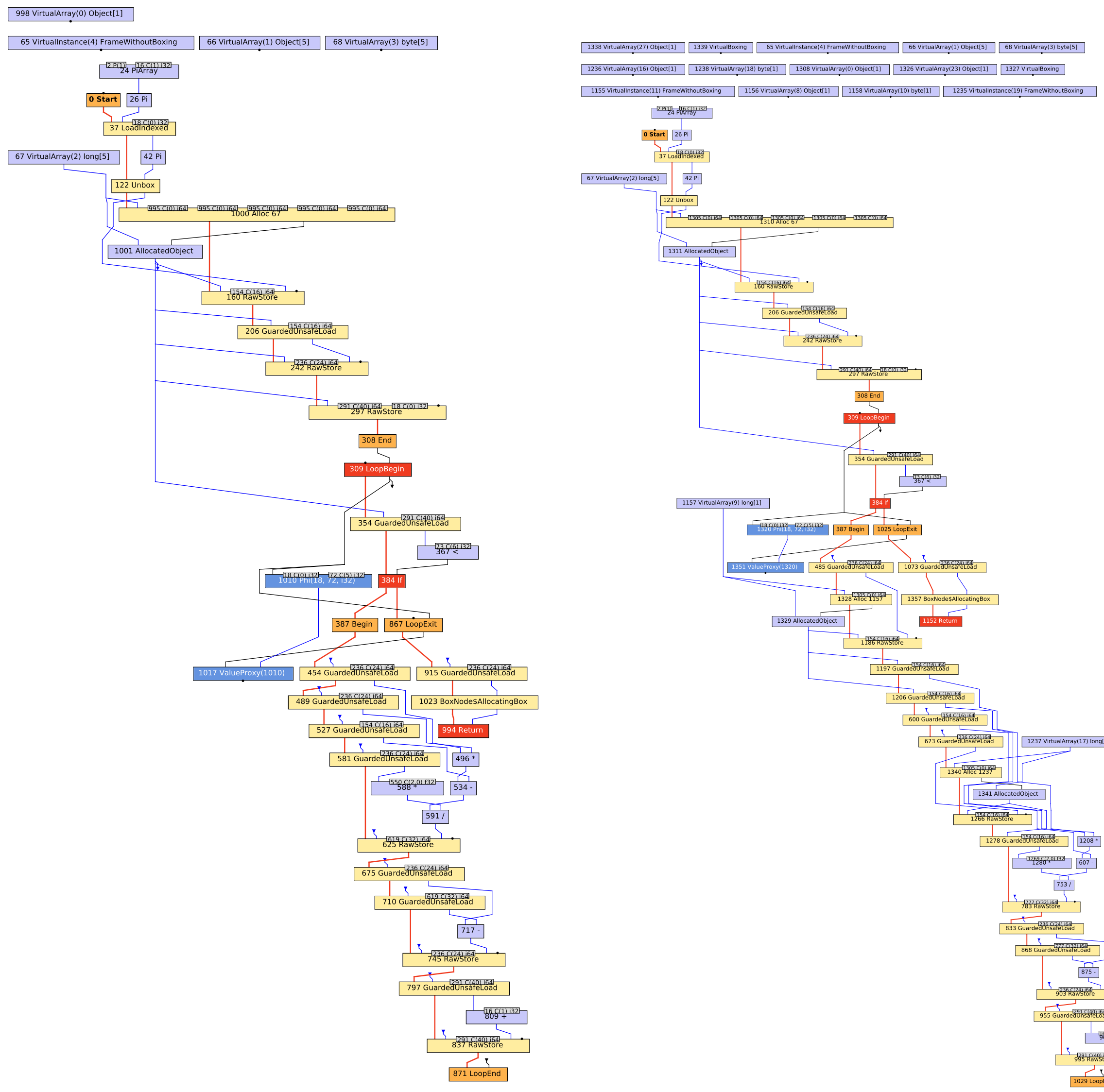


Figure 6.2: Difference in Graal intermediate representation (IR) between square root with inlined (left) and non-inlined (right) function and derivative.

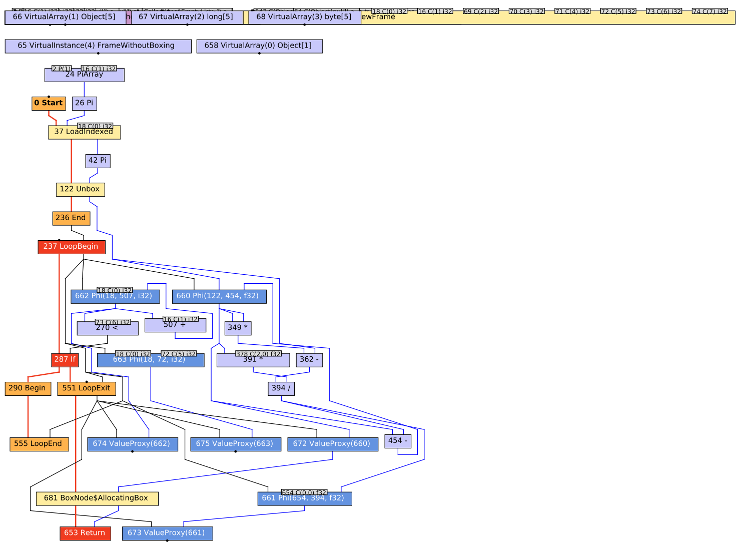


Figure 6.3: Graal IR for square root after the issue with the variable data types has been fixed. The source code is the same as in Figure 6.2 on the right.

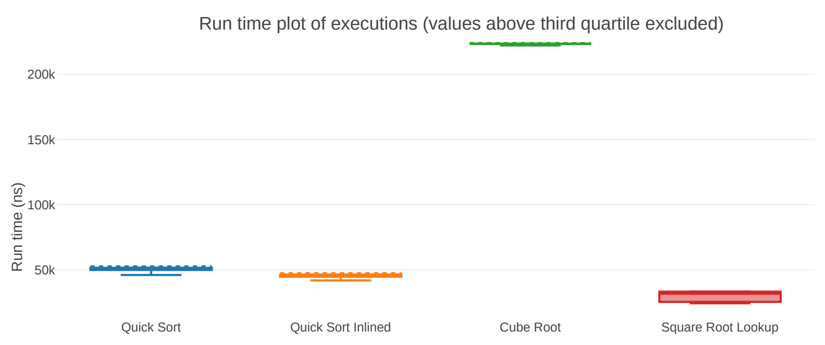


Figure 6.4: Fixed write nodes show a significant performance improvement for all algorithms.

this bug. Figure 6.3 shows the Graal IR after this fix has been applied for the entire MiniC Language. Only one IR is shown, as the graph is now identical for both versions of square root.

This discovery has several consequences for our work. On the one hand, it already shows the successful application of IGOR to identify patterns that impact the run-time performance of software. On the other hand, it also shows its limitations. The identified pattern is based on the *genotype*, e.g. the AST, of the source code. Graal however works with the *phenotype*, the Graal IR of the source code. While applying the anti-pattern *invoke*, would have been possible with a mutation operation in KGGI that inlines functions, which would have brought a significant performance improvement to many of the algorithms in the test set, the true cause of the performance loss, the missing metadata for the compiler in the frame slots, would be entirely ignored. This fix improves the performance even further than the pattern could have, but required manual investigation of the root cause. Figure 6.4 shows the new performance of the algorithms after applying the fix to MiniC.

Performance of the Manually Written ASTs

All the experiment suites were benchmarked on one input, over 200,000 executions, of which the first 100,000 executions are discarded. Each individual benchmark was done in a separate JVM using the Amaru framework (see Section 10.3) developed as part of this work. To ensure comparability, all performance measurements are given in nanoseconds in the entire chapter.

```
float cbrt_benchmark(float x) {
    int i;
    float sum;
    sum = 0.0;
    for (i = 0; i < 100; i = i + 1)  {
        sum = sum + cbrt(x+i) + cbrt(x+i+1) +
cbrt(x+i+2) + cbrt(x+i+3) + cbrt(x+i+4) + cbrt(x+i+5)
+ cbrt(x+i+6) + cbrt(x+i+7) + cbrt(x+i+8) +
cbrt(x+i+9);
    }
    return sum;
}
```

Listing 6.2: Example of how the math functions were benchmarked. Shown is the cube root benchmark

The benchmark for the math functions was performed on the input 0.5 for all functions. As the math functions execute too quickly for a

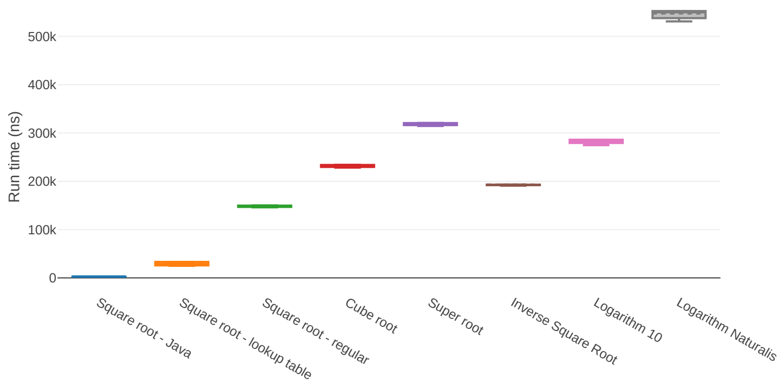


Figure 6.5: Performance of the manually written math ASTs.

single benchmark call, the benchmark instead uses 1,000 *in-process* calls, meaning that a benchmark function is called in the MiniC guest language, which performs the 1,000 calls. To ensure that Truffle and Graal do not optimize these calls away, or reduce them, 100 loops are conducted which add the sum of 10 calls with incrementing input values. The benchmark for the cube root function is shown in Listing 6.2.

Figure 6.5 shows the run-time for all algorithms in the suite. The access to the square root implementation of java outperforms the lookup table implementation, which in turn outperforms its pure Newton-Raphson version. Since the Cube root and Super root implementations are just increasingly complex adaptions of square root, their run-time is as expected. Inverse Square root has a similar run time to regular square root, which is expected, as the only difference is one division operation. Logarithm 10 and Logarithm Naturalis have similar implementations, only differing in the constant applied. It is likely that more steps are required to reach the target accuracy threshold in Logarithm Naturalis.

The sorting algorithms were benchmarked on the same array with a size of 1,000. As this already shows sufficient differences in performance, no in-process interactions were used. In Figure 6.6 Bubble sort, Insertion sort and Selection sort behave as expected, considering that their average run-time complexity is $\Theta(n^2)$. Shell sort ($\Theta(n(logn^2))$) and Shaker sort ($\Theta(n^2)$) however have an unusually low run-time. Heap sort and Merge

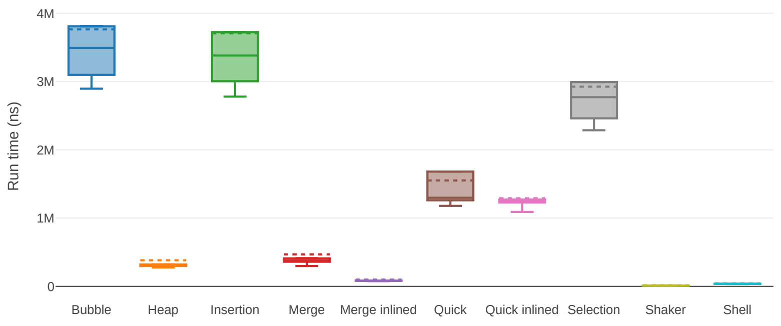


Figure 6.6: Performance of the manually written sort ASTs.

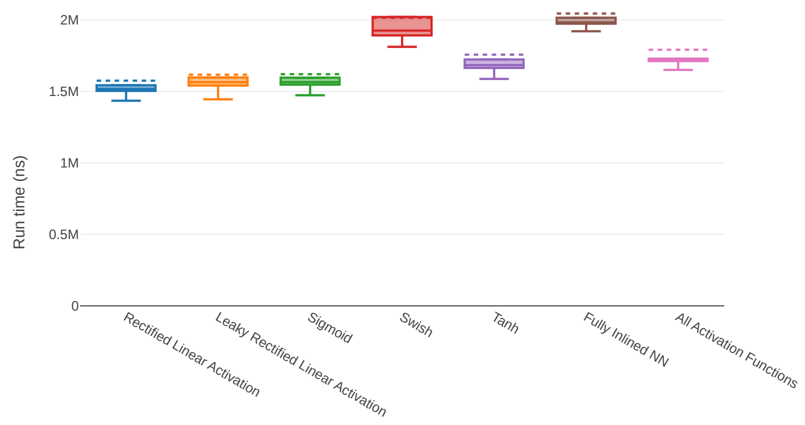


Figure 6.7: Performance of the manually written neural network ASTs.

sort ($\Theta(n(log n))$) behave as expected, with Quick sort ($\Theta(n(log n))$) being slightly less performant than expected. Both inlined variants of Merge and Quick sort have a better runtime than their counterparts not using function calls.

The neural network algorithms were benchmarked on a test suite producing an XOR gate. Figure 6.7 shows the performance of the respective implementations. Rectified Linear Activation and its Linear variant, as well as Sigmoid, have the best run-time performance. Tanh, as well as the version with all activation functions also have a similar run time. This is interesting, since the version with all activation functions uses swish as a function call instead of having it inlined, making it functionally equivalent to the version using Swish. This suggests that the Graal compiler can apply different optimizations while the function is not inlined. The fully inlined neural network having the longest runtime seems to confirm this.

6.2 Experiment Baseline - KGGI Without Patterns

To thoroughly evaluate the results of applying patterns in KGGI a baseline is established. In this experiment, all algorithms described in the previous section are modified via KGGI *without any patterns*. The goal of these experiments is to showcase how a genetic algorithm would fare in the search space of MiniC, and enable a comparison to the influence of applying patterns to the search space.

To ensure reproducibility of the results, all 25 experiments utilize the same parameters and set up:

Search Space Almost the entire language of MiniC is selected, the only exception being the *read from console* built-in function. This function is omitted in all tests, as it would disproportionally block any AST since the tests are not interactive. This amounts to *108 operators* and *59 operands*, totaling 167 options.

Initial population creation The initial population is created via mutants of the original AST.

Crossover A single-point crossover is chosen. As KGGI considers the validity of nodes according to the grammar of MiniC, the crossover always creates compilable individuals. In addition, it has a strict depth and width restriction, where depth means the distance between the root node and the furthest leaf, and width means the maximum amount of direct children that one node may have. The limit is different for every experiment, with a depth and width allowed to exceed the original solution by one.

Selection A tournament selection is applied with a tournament size of 10. In addition, the selector only considers such ASTs for selection that do not result in run-time exceptions, unless no AST runs without exception. This was done, since not excluding such ASTs leads to experiment runs with almost 100% of individuals producing run-time exceptions.

Elitism The best individual from the population is kept in the population.

Mutator The mutator is a single-point random mutator. Similar to the crossover, it only creates mutants that are able to compile, and the same depth and width restriction applies. In addition, to ensure that the mutator does not create too large individuals, the width, and depth limits are randomly reduced at the mutation point. For example, if a mutation point close to the root is chosen, the mutator is randomly assigned a maximum depth of between 3 and 11, instead of a fixed maximum depth of 11. This is done as the random mutation in MiniC tends to create large sub-ASTs. The mutation probability is 13%. While this is a high rate for GP, it is quite low compared to current GI frameworks, which use local search algorithms exclusively using mutation [70, 107–109].

Population size and Generations A population size of 100 with a generational limit of 20 is selected. This enables a generation of approximately 2,000 ASTs per sub-experiment, totaling approximately 50,000 ASTs over the entire experiment.

Fitness Function The fitness function is the accuracy on the test cases as defined in Algorithm 1. Every successful test case where the expected output matches is rated 0, while tests that do not match the output are rated between 0 and 1 depending on how far the output is from the expectation. Returning the wrong data type is ranked 2 and a run-time exception is ranked 10.

Timeout Every AST has a chance to never finish its execution or produce an unrecoverable run-time exception. For example, an Out of Memory exception crashes the Java Virtual Machine (JVM) without the ability to catch the exception. A timeout was introduced so that each AST has as upper limit until it is considered failed. For the *math* and *sort* experiment suites, the timeout was set to 3 seconds. For the *nn* experiment suite, the timeout was set to 10 seconds.

Over the entire experiment, 15,906 different solutions were created. 4,494 of them succeed every test case, 9,470 produce a run-time exception in at least one test case, and 1,942 produce invalid results, either returning the wrong data type, or not the exact value required by one or more test cases. In total, all AST consist of 10,369,142 nodes.

The experiment performed 40,967 crossover, 6,058 mutation and 2,525

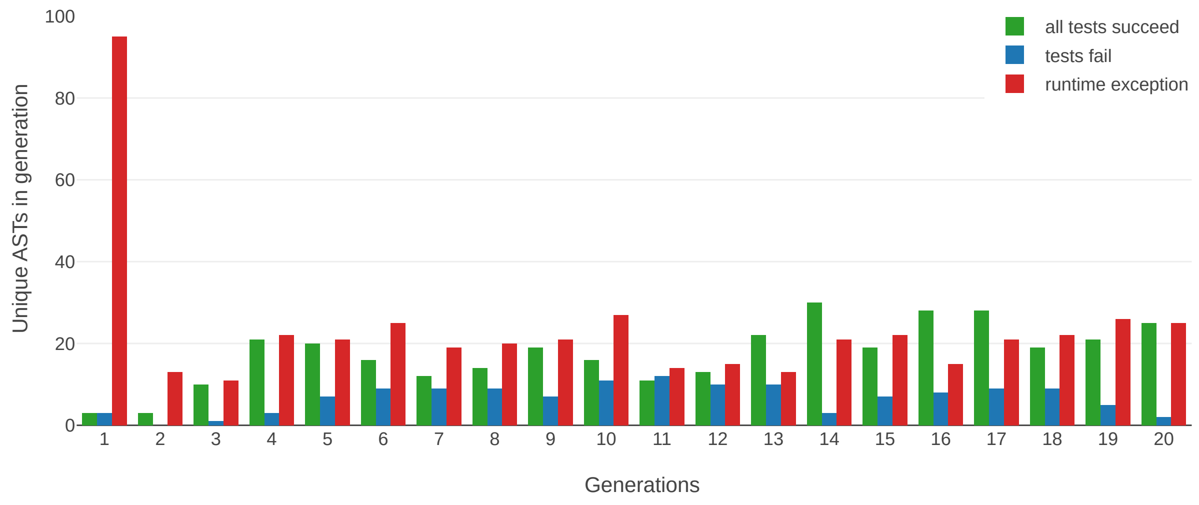


Figure 6.8: Histogram of unique solutions per generation in the KGGI experiment run of *shaker sort*. Uniqueness is considered in the generation, not overall. The groups are *individuals succeeding all tests*, *failing at least one test*, or producing a *run-time exception* in at least one test.

create operations, for a total of 49,550 operations. 450 less than 50,000 due to elitism over the generations following the first generation. The expected number of approximately 50,000 ASTs is not reached. The diversity of the AST over all experiments results in every solution on average being present 3.14 times over all populations and generations. Considering the large search space, this shows a severe lack of diversity in the populations. The reason for this can be seen in Figure 6.8, which shows a histogram of all unique solutions per generation in the KGGI experiment run on shaker sort. The first generation is created purely via mutation, resulting in 100 different ASTs. The first generation produces only 6 individuals that do not produce run-time exceptions, 3 of which succeed all test cases. While all later generations, which rely on crossover with only a 13% mutation chance, produce far fewer failing individuals, there is far less diversity in the generations than expected.

Table 6.4 shows how likely each operation is to create successful individuals. It should also be noted that many individuals repeatedly occur in multiple generations, so the overall diversity is even less than visualized in Figure 6.8, for example in shaker sort a total of 645 unique ASTs occured over all generations. This behavior is similar for all 25 experiment runs, with some notable differences:

- In the *math* experiment suite some individuals also return the

	Successful	**Failed test**	**Exception**	**Total**
Create	128 (5,1%)	117 (4,6%)	2,280 (90,3%)	2,525
Mutate	820 (13,5%)	290 (4,8%)	4,948 (81,7%)	6,058
Crossover	27,698 (64,8%)	6,271 (14,6%)	8,796 (20,6%)	42,765

Table 6.4: Operation success rates in KGGI without patterns. Successful is only counted if all test cases are succeeded exactly, or in the case of neural networks if every test-value is within a 1% margin. If at least one test case fails, ASTs are grouped in *failed test*, and if at least one run-time exception occurs, ASTs are grouped by *run-time exception*

Table 6.5: Unique AST individuals in math experiment baseline of KGGI.

Experiment	Successful	Failed test	Run-time Exception	Total
Square root - Java	278 (47.4%)	23 (3.9%)	286 (48.7%)	587
Square root - lookup table	164 (26,8%)	45 (7,4%)	403 (65,8%)	612
Square root - regular	374 (44,3%)	47 (5,6%)	423 (50,1%)	844
Cube root	347 (44,4%)	98 (12,6%)	336 (43%)	781
Super Root	1 (0,2%)	205 (35,9%)	365 (63,9%)	571
Inverse Square Root	238 (30,9%)	105 (13,6%)	428 (55,5%)	771
Logarithm 10	238 (34,1%)	15 (2,1%)	445 (63,75%)	698
Logarithm Naturalis	150 (27%)	18 (3,2%)	388 (69,8%)	556
Total	**1,790 (33%)**	**556 (10,3%)**	**3,074 (56,7%)**	**5,420**

wrong data type.

- in the *sort* experiment suite, the diversity grows after the first few generations, especially *selection sort* and *merge sort inlined*, which in one generation contains 75 different successful ASTs.
- The *neural network* experiment suite generally has the lowest diversity. While overall quality tends to improve over time, the amount of successful individuals also tends to decrease, with tanh producing no successful individuals in the last three generations.

In the math experiment suite, as shown in Table 6.5, 5,420 out of approximately 16,000 expected unique ASTs are generated. Of these, more than half produce an exception in at least one test case, and about one third generates a successful solution. The super root has only one successful individual in its population, but also the highest rate of individuals that produce a result that does not match the desired test result. This may indicate that math functions with a higher complexity are harder to produce, which is also confirmed by the low success rates in inverse square root, and the two logarithms. The square root - lookup table might be an outlier or alternatively be more susceptible to failures since it has invocations to three different functions.

The sort experiment suite (see Table 6.6), shows a similar overall behavior, even though the suite has ASTs that are much larger than in the math functions. With 6,733 out of an expected 20,000 AST it has a similar diversity as the math functions and it's distribution of successful to failed tests and runs with exceptions is almost identical. Heap sort and quick sort inlined, which are the two largest ASTs also are the least successful

Table 6.6: Unique AST individuals in sort experiment baseline of KGGI.

Experiment	Successful	Failed test	Run-time Exception	Total
Bubble	263 (36,9%)	74 (10,4%)	376 (52,7%)	713
Heap	92 (15%)	146 (32,8%)	376 (61,2%)	614
Insertion	125 (31,7%)	58 (10%)	394 (68,3%)	577
Merge	217 (33,1%)	80 (12,2%)	359 (54,7%)	656
Merge Inlined	510 (55,3%)	89 (9,7%)	323 (35%)	922
Quick	173 (27%)	14 (2,2%)	454 (70,8%)	641
Quick Inlined	113 (21,7%)	19 (3,6%)	390 (74,7%)	522
Selection	308 (41,5%)	85 (11,5%)	349 (47%)	742
Shaker	152 (23,6%)	83 (12,9%)	410 (63,5%)	645
Shell	281 (37,9%)	62 (8,4%)	398 (53,7%)	741
Total	**2,234 (33%)**	**710 (10,5%)**	**3,829 (56,5%)**	**6,773**

Table 6.7: Unique AST individuals in neural network experiment baseline of KGGI.

Experiment	Successful	Failed test	Run-time Exception	Total
Rectified Linear Activation	65 (12,3%)	106 (20%)	358 (67,7%)	529
Leaky Rectified Linear Activation	94 (17%)	74 (13,4%)	385 (69,6%)	553
Sigmoid	0 (0%)	136 (27%)	368 (73%)	504
Swish	96 (19%)	62 (12,3%)	347 (68,7%)	505
Tanh	8 (1,5%)	162 (29,3%)	382 (69,2%)	552
Fully Inlined NN	107 (19,4%)	75 (13,6%)	369 (67%)	551
All Activation Functions	100 (19,3%)	61 (11,7%)	358 (69%)	519
Total	**470 (12,7%)**	**676 (18,2%)**	**2,567 (69,1%)**	**3,713**

runs. Merge sort inlined, however, which is the third largest AST is the most successful run with more than half of all individuals succeeding.

Finally, the neural network suite shown in Table 6.7, produces 3,713 out of 14,000 expected ASTs. Instead of averaging 56% of ASTs with run-time exceptions, the failure rate is much higher with 69.1%. This may be due to the much larger AST size than in the other experiments. The rate of tests that do not produce the desired result of the target output within a 1% margin is also higher than in the other experiment suites. Sigmoid has not a single individual that produces the desired result.

Over all experiments, the baseline for KGGI shows a low diversity in the populations as well as a high rate of ASTs producing exceptions. However, all experiments, except the super root, produce multiple successful AST solutions as a baseline. All experiments show a tendency to either fulfill all test cases, or produce run-time exceptions. The fact that over most experiments, the test cases that fail but do not produce run-time exceptions is the smallest group is noteworthy.

6.3 Mining of Mutational Bug Patterns

As a first attempt to answer how recurring patterns that impact and improve a functional or NFP can be identified, the exceptions occurring due to incorrect mutations in ASTs during the baseline experiments are analyzed. Depending on the context, these could be considered functional patterns, since these mutational bugs negatively impact achieving the desired test output. In GI however, this is considered as a NFP - *correctness*.

This experiment is done using the IGOR algorithm as outlined in Section 4.6. Due to the large search space size of 9,470 ASTs that produce at least one run-time exception per test case, it is likely that IGOR would only identify the most common issues, which would show the highest discriminative threshold, and likely have many similar patterns growing around the core.

Thus, the experiment is split into the different exception classes that occurred during the baseline experiments. Table 6.8 shows all 12 classes in descending order by occurrence that were analyzed via IGOR. Figure

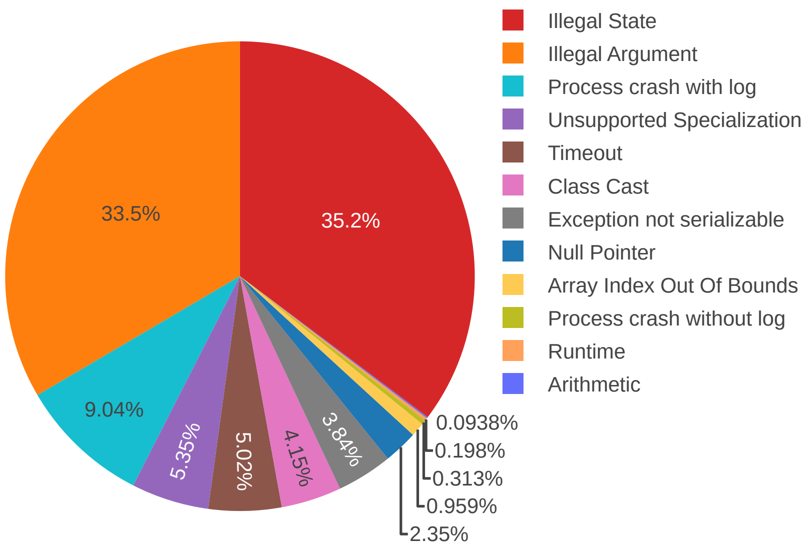


Figure 6.9: Distribution of the exception groups in percent according to the occurrences in Table 6.8

6.9 shows the percentage that the bug class takes over all exceptions. It should be noted that several ASTs had different run-time exceptions in different tests. For example, almost all exception types occur together with the *Process crash with log* exception. Most of the exceptions are well-known exceptions from the JVM. The *Unsupported Specialization Exception* is unique to the Truffle framework and occurs if an unexpected node is child of another node. This can happen if a constructor for a Truffle node is more generalized than the context would actually demand during run-time. The *Process crash with log* exception means that there is no known exception, but rather the Amaru (see Chapter 10) worker that attempted to execute the AST crashed while still successfully writing a log file. The *Process crash without log* exception is the same, but no log file is available. Finally, the *Exception not serializable* class occurs if the AST execution failed, but the reason could not be sent back from the executor service.

The mining process for every exception class is identical and actually happens four times, to analyze the effects of *embedded* vs. *induced* mining as well as the effects of the hierarchical mining inherent to IGOR:

Induced Mining The induced mining uses the entire population of

Table 6.8: Overview of the number of **bugs** that occur over all experiments with KGGI. Occurrences are counted per unique AST. Different test cases may return different exceptions, thus ASTs may occur in multiple groups.

Exception	ASTs	Nodes	Relationships
Illegal State	3,380	847,015	843,713
Illegal Argument	3,208	890,365	887,156
Process crash with log	867	171,718	170,851
Unsupported Specialization	513	134,468	133,955
Timeout	481	116,387	115,906
Class Cast	398	152,101	151,703
Exception not serializable	368	126,010	125,642
Null Pointer	225	60,445	60,220
Array Index Out Of Bounds	92	12,140	12,048
Process crash without log	30	4,738	4,708
Runtime	19	2,252	2,233
Arithmetic	9	1,703	1,694
Total	**9,590**	**-**	**-**
Successful	4,494	593,367	589,338

ASTs per group compared to the entire successful population. The induced mining is done once including the literal values of the nodes, and once without the literal values, only retaining the normalized variables.

Embedded Mining The embedded mining does not utilize the entire population due to memory limitations. Table 6.8 gives an overview of the amount of nodes and relationships in the corresponding search spaces. The available memory of 120 GB RAM was exceeded when attempting the runs on the entire population and ultimately had to be reduced to 15 individuals per group in the embedded space.

What is identical for all four execution settings is:

Pattern size is limited to 5,000. As all AST are smaller this means that there is technically no upper limit.

Hierarchy is the default MiniC hierarchy as given by its implementation, concentrating primarily on the data types.

Redundancy the strategy for redundancy reduction is *closed*, meaning only the largest pattern containing sub-patterns survives.

Metric An expansion of the *maximal contrast* metric was used. In addition to ranking all patterns by their contrast, the metric demands that no outliers exist. This means that patterns occurring in the exception class were only relevant if the pattern did not occur at all in the successful group.

Faults vs. Faults of omission The mining was applied twice to identify co-located patterns. First, the metric was applied to maximize the discriminability in the exception class, and demands no outliers there. Then this was inverted to identify patterns in the successful group that had no corresponding outliers in the faulty group.

Growth For the *induced* mining the 200 most discriminative patterns were grown every time the pattern size was increased. Again due to memory limitations this had to be reduced to 30 for the *embedded* mining.

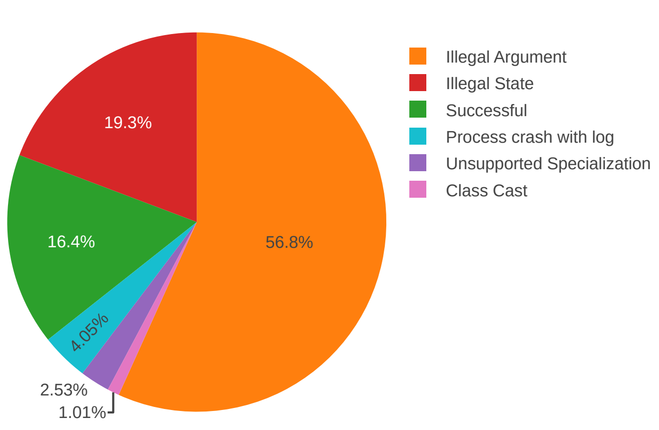
Top N return Only the top 10 patterns according to the discriminability metric were recorded in the final report.

Pattern Verification

After the pattern mining was conducted, the patterns as identified in the reports were manually analyzed. They were then transformed into machine-readable anti-patterns, responsible for a bug occurring and additional anti-patterns to be prevented in future runs of KGGI. For bugs that could not be prevented outright positive patterns were introduced.

All anti-patterns and patterns were analyzed via the verification mechanism outlined in Section 5.3. This analysis always consists of first attempting to validate the reason why an exception class occurred, e.g., the anti-pattern, and then applying a fix (e.g., anti-pattern prevention or pattern induction) in a second verification. Both verifications are based on the mutation operation of KGGI.

A specialized mutator is introduced that allows enforcing a specific mutation point, e.g., any point where the anti-pattern can actually be validly introduced. The mutator only conducts one mutation to ensure



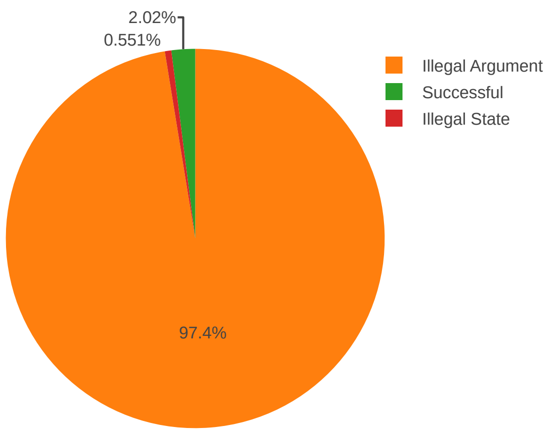


Figure 6.10: Example of pattern verification with mutated ASTs on the left, and original ASTs on the right. Shown is an attempt to prove an anti-pattern responsible for the *Illegal Argument* group. When utilizing the ASTs modified by KGGI the confidence is only 56.8% as opposed to 97.4% confidence in the pattern when utilizing the original ASTs, since the mutations have a large amount of dead branches.

that the anti-pattern or pattern are applied without side effects from other mutations. This has to happen since ASTs when mutated may actually produce a different bug than the one that should be identified by the anti-pattern. In addition, the verification process is conducted only on the original 25 ASTs of the experiment suite. The reason for not attempting to introduce the exception in the valid solutions created in the baseline can be seen in Figure 6.10. It shows the same anti-pattern verification on the exception class of *Illegal Argument*, on the left with all valid solutions and on the right with only the original manually written ones. The solutions produced in the baseline have a large amount of dead branches that never execute, severely impacting the ability to analyze patterns from them. On average, the confidence in proving anti-patterns drops by 40% when not utilizing the original AST solutions. This issue with dead branches is also a hindrance for the pattern mining itself, as successful individuals can have bug patterns in dead branches that do not impact the solution. In the future the approach could be expanded to only consider branches that are executed for mutation.

Every verification is done by conducting 100 mutations on randomly selected ASTs, since only 25 ASTs are available this means that on average every pattern is tested four times per AST. The confidence in a pattern is calculated by how often the exception occurs per test case that is available for the given ASTs, instead of just attempting one test case per AST. This is done to verify a pattern over different branches that may be taken during different test cases. When attempting to validate the fix for an anti-pattern, the confidence is instead calculated by how often the exception does not occur. For example, 98.1% confidence means that the exception occurred in 1.9% of all tests.

Identified Patterns

In what follows, every exception group is discussed concerning their identified anti-patterns and implemented fixes for them, grouped by the cause of the issue. Table 6.9 gives an overview which type of mining identified a bug pattern. For every discussed group, the confidence is given. As every pattern is verified individually, this confidence is sometimes influenced by bugs which are caused by similar issues. In

Table 6.9: Overview over which pattern was identified via which type of mining. Overall, both induced and embedded mining find different anti-patterns. Embedded mining identifies patterns more reliably than induced mining. If normalization is conducted or not seems to make little difference as well.

Exception	**Induced**		**Embedded**		**Notes**
	Literals	Normalized	Literals	Normalized	
Illegal State		x	x	x	Fault of Omission only found in Embedded
Illegal Argument				x	
Process crash with log					Reason not in ASTs
Unsupported Specialization			x		
Timeout					No provable patterns found.
Class Cast	x		x	x	Induced and Embedded with Literals find different pattern than Embedded Normalized
Exception not serializable	x	x			
Null Pointer			x	x	
Array Index Out Of Bounds	x	x	x	x	
Process crash without log					Reason not in ASTs
Runtime		x			
Arithmetic	x	x	x	x	

the figures of this section, patterns in *red*, are the cause of a bug. *Green* patterns attempt to influence the search space positively to reduce the bug and *blue* anti-patterns restricting the search space to prevent the bug. In two instances, instead of creating new patterns a strategy was introduced in KGGI that deals with the issue instead, shown in *gray*.

Data-Flow-Related Bug Patterns

Most identified patterns are related to data flow. This includes *Illegal State*, *Illegal Argument*, *Null Pointer*, and some patterns of *Class cast*.

Illegal State is being caused by the only identified fault of omission. Figure 6.11 shows the cause, which is a *read* without a corresponding *write* to the same variable (82.7% confidence). This means that an uninitialized stack or heap variable is being read, causing the issue. The fix is to require a *write* to a variable before a read becomes valid (94.3% confidence). This fix has been applied in a new *data flow strategy* introduced in KGGI which also handles the other issues related to data flow. This is done primarily as underlying bugs together would consist of more than 40 anti-patterns

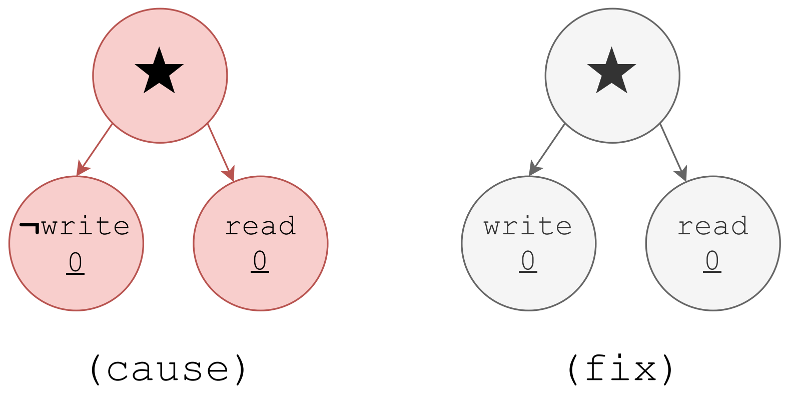


Figure 6.11: Pattern causing the Illegal State exception when a *read* happens before the variable was initialized with a *write* (left). A corresponding fix is only allowing a *read* to variables with a preceding *write* (right).

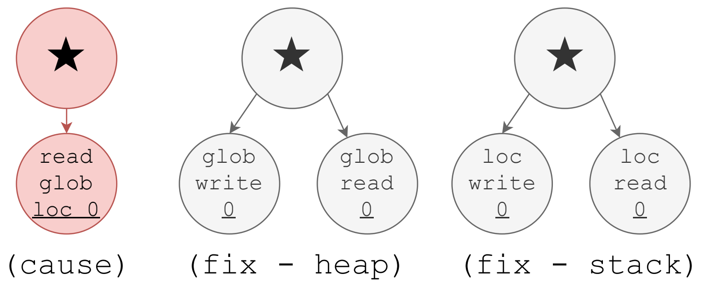


Figure 6.12: An Illegal Argument exception happens whenever a *heap read* happens on a stack variable (left). The fix is the separation of stack and heap *reads* (right).

(each data type, heap vs. stack, chaining of data flow operations, ...), which becomes more easily manageable as a single strategy conducting this verification.

Illegal Argument is caused whenever a *heap read* is attempted on a variable existing only on the stack but not the heap (97.4% confidence). The fix is to extend the *data flow strategy* to consider which variables the *read* and *write* nodes have access to (100% confidence). Since Truffle does not restrict the implementation from having multiple materialized frames (e.g. multiple separate heaps) the implementation was changed to be frame-specific to allow extensibility.

Null Pointer happens on three types of read (see Figure 6.13). *char read* (77% confidence), *string read* (77.7% confidence) and *array read* (85.9% confidence) on an initialized variable cause a *Null Pointer* exception instead of an Illegal State exception. If the variable is initialized and of a different data type, the exception instead is Illegal State which is the second most occurring issue. The reason why this results in a null-pointer exception is due to how variable access is designed in Truffle. Since Truffle is a framework for designing languages in the JVM the primitive

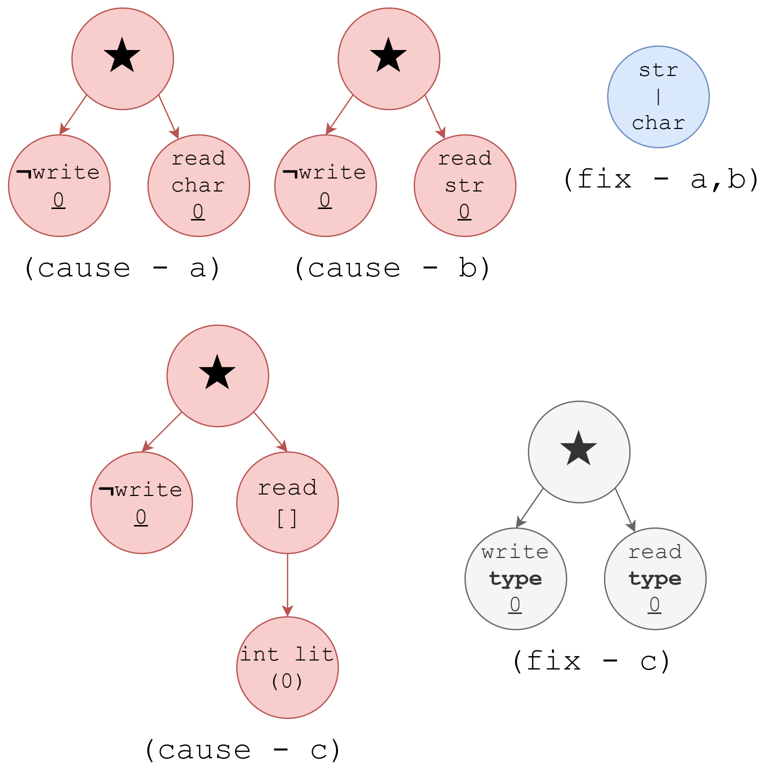


Figure 6.13: A Null Pointer exception happens only on *reads* of the data types char, string and array (left). The fix is to only allow reads with a corresponding *write* (right). In the case of string and char, the data types were prevented completely via anti-pattern since no experiment in the entire suite uses the data types.

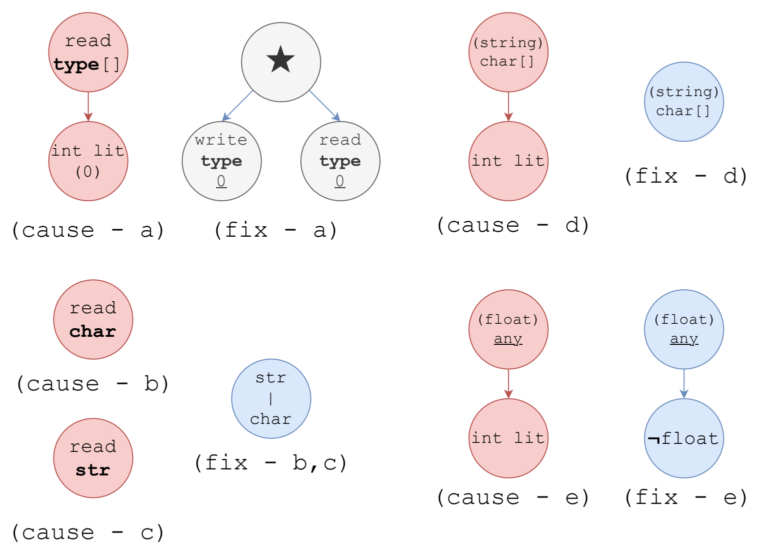


Figure 6.14: Class Cast has the most patterns identified causing it. One reason is a typed read on a variable that has a different object data type written to it. This can be seen in a) on an array reading a different array type (e.g. int array reads double array), a char reading a string or vice versa, or both data types reading an array. The fix is to only allow reads on variables with a matching preceding type. The other two causes are because of cast nodes (d, e) with a solution to enforce only the correct source data type being cast.

data types will not always match the data types of the designed guest languages. Such data types, in the case of MiniC char, string and arrays, are represented as a generic Object type. As the frame slot type of *null* cannot be verified, the Illegal State exception is not caused, resulting in a *Null Pointer* exception whenever the non-existing value is being transformed into the guest language's primitive data type.

Unfortunately, for the three given causes of *Null Pointer* only the fix for *array read* can be proven (97.9% confidence fix - c). The fix is already sufficient since an *Illegal State* exception is already prevented via the *data flow strategy*. The reason why *char* and *string* can't be proven shows a limitation of our approach. No experiment in the suite uses these two data types. This would require two mutations in the available experiment space, one to add a *write* and one to add a *read*, instead of only one as designed for the verification. Since not a single experiment uses the string and char data types, they were excluded entirely from the search space via an introduced anti-pattern.

The final group is *Class Cast* which is partially caused because of data flow. Figure 6.14 shows causes a-c which relate to data flow. Similar to *Null Pointer* the cause is any node with an underlying object type in Truffle reading from a variable. In this case, the variable is set to a different object type. If, for example, a *read-int-array* node attempts a read of a double array or a read-int-array node attempts to read a string or char, this causes a class cast exception instead (93.5% confidence). The same holds for a *read-char* node attempting to read a string or any array type (94.1% confidence) or a *read-string* node attempting to read a char or array (95.8%) confidence. The fix for this class cast issue, is to extend the *data flow strategy* to enforce an order on any node accessing the data flow. Valid orderings go beyond the identified patterns and include for example *(allocate int array)*→*(write int array)*→*(read int array)*. This is not manually done, but instead conducted by mining the language once, and deriving valid pairings as information applied in the *data flow strategy*. Section 10.2 describes in more detail how the language analysis is conducted. This approach has been proven with the *array-read* pattern (85% confidence). It should also solve the char and string *read* patterns, which are not provable due to the approaches' limitation.

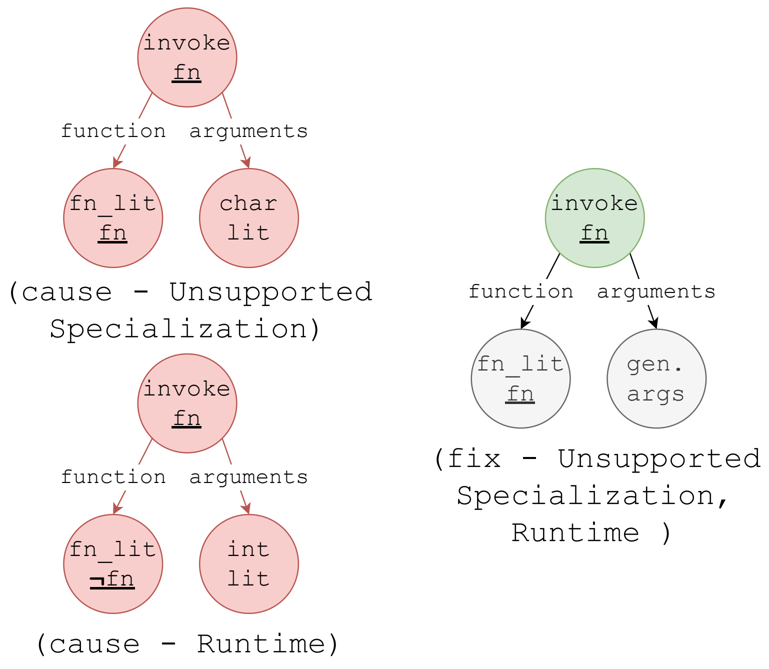


Figure 6.15: Unsupported Specialization is caused by a function being invoked with an incorrect argument type (left top). Runtime exceptions are caused whenever a function is called that does not exist (left bottom). Both issues are fixed by synthesizing invocations of all functions in the function registry.

Ambiguous Language Implementation Bug Patterns

Three types of patterns are caused by how languages in Truffle are designed, and may be specific to MiniC itself rather than generalizable over multiple languages. This includes *Unsupported Specialization*, *Runtime Exception*, and the remaining two patterns of *Class Cast*.

Unsupported Specialization as well as *Runtime* are both closely related, and caused whenever invocations of other functions are attempted incorrectly. *Unsupported Specialization* is caused by a function being called with incorrect arguments (96.7% confidence). This is proven by injecting a char as a function parameter, which will always cause the *Unsupported Specialization*, since no AST in the experiment employs char nodes, as shown in Figure 6.15. The root cause is an ambiguity, since function dispatches need to be implemented generically. In the AST any node returning a value is acceptable.

A *Runtime* exception is thrown instead when a function is attempted to be called that does not exist. This is caused by an ambiguity in how MiniC, and Truffle languages in general, are designed. The invocation is done by a dispatch to another function. Which function is usually decided by the parser or linker, which correctly injects functions. In the AST the function node is simply a reference to the function, in the case of MiniC this is a string literal referencing the correct node in the function registry. Thus, the AST can be generated with non-existing function references.

For both issues, the same fix pattern can be applied. A new *invocation strategy* synthesizes patterns for every function available in MiniC's function registry. The synthesis is conducted for each return type, and consists of the function name, as well as the arguments by collecting the argument read nodes in the AST of the function to collect the data types. This approach can be improved in the future, as some read nodes are generic and can't exactly determine the underlying data type. For example, only one generic node exists for arrays, independent of the

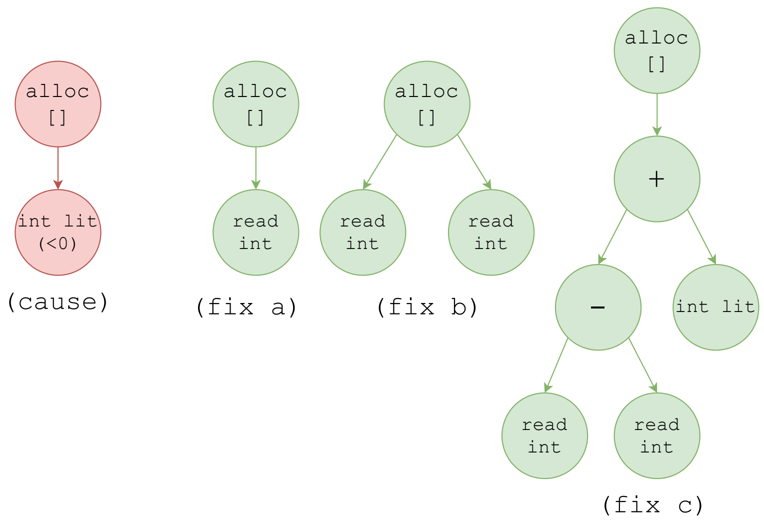


Figure 6.16: Exception not serializable is caused by at least one identified pattern. Whenever an array is allocated with a negative size (left), the AST fails at run time. No corresponding pattern to prevent this was identified, and thus three positive patterns were introduced to guide the search space into a more sensible direction (right three patterns).

underlying array data type. The synthesized patterns fix both *Unsupported Specialization* (99.3% confidence) and *Runtime* (100% confidence).

Two patterns in the *Class Cast* group are caused by the same issue that is causing *Unsupported Specialization* and *Runtime*. The general issue is that some Truffle nodes allow a more general data type in their interfaces than is allowed at run time. This anti-pattern is most easily identified by any node accepting any node type instead of a more specific type. A manual search through MiniC was conducted and resulted in the only two patterns not identified via pattern mining. Figure 6.14 shows two nodes conducting casts, but accepting invalid child nodes that they cannot cast. The cast of a char-array to string accepts any type, since arrays are only identifiable as object during the parse process (d in Figure 6.14). The anti-pattern is proven by injecting an int literal (97.1% confidence). An acceptable fix would be to only allow nodes returning the correct the data type, similar to how the *data flow strategy* matches data types. However, this pattern is once again not provable, and the node is excluded from the search space. The second cast (e in Figure 6.14) is a specialized node for unboxing from Java. It accepts any object but can only cast the float boxed data type. Other specializations of the same node could actually parse an int, but cannot be used if the specialized node is injected instead, which is why the anti-pattern can be proven (97.4% confidence). The fix is to only accept nodes returning float values as child nodes (90.2% confidence).

Data-Type-Specific Bug Patterns

The two classes *Exception not serializable*, and *Array Index Out Of Bounds* are both caused by incorrect array handling.

Exception not serializable is caused by a bug in the transmission between workers of the Amaru framework, as was later discovered. Even though this obfuscated the actual bug message for mining, at least one pattern causing this was still discovered. Whenever an array of any type is allocated with a negative size (see Figure 6.16), the AST fails at run time (97.3% confidence). This is the first of two instances where a part of the presented pattern was not discovered via mining. Only the *allocate* was

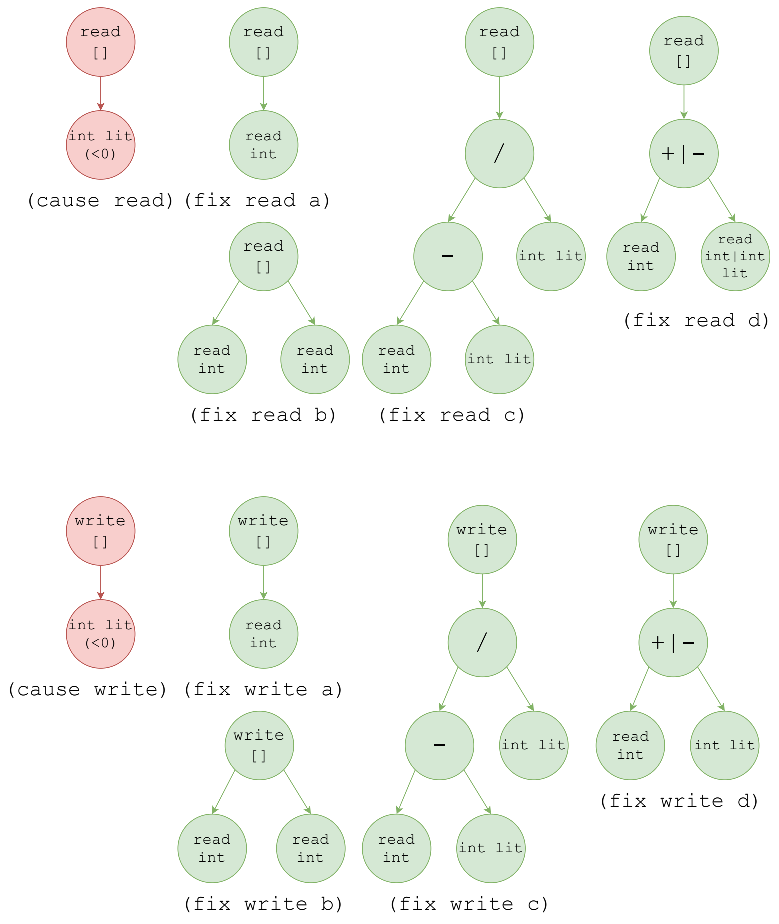


Figure 6.17: An Array Index Out Of Bounds exception happens when read (top left), or write (bottom left) happens outside the array bounds. This was proven with a negative read. Positive patterns were identified to guide the search space for read (right patterns top), and write (right patterns bottom).

discovered, and the negative array size was discovered when analyzing an AST that produced the exception during validation. The reason is, that not a single AST fails because of a negative literal. Instead, sub-ASTs, conduct various math operations, casts and function calls that result in a negative value. The way the negative value is reached is so diverse that no discriminative pattern was found. To guide the search space in a better direction, the succeeding ASTs were mined for positive patterns, which resulted in similarly counter-intuitive sub-ASTs. Thus, the original 25 ASTs were mined for patterns allocating arrays instead. The result are the three positive patterns shown in Figure 6.16), which allocate a one-dimensional array (a), a two-dimensional array (b), or a single dimensional array by subtracting two variables from each other and adding a constant (c). The allocations are independent of any array data type (97% confidence). However, this solution does not guarantee that the bug will not occur anymore, since any variable being read for allocation that has a negative value will still cause this bug.

Array Index Out Of Bounds is caused by two very similar issues. Instead of an issue with the allocation, when an allocated array position is being read out of bounds (79.8%) or a position is being written out of bounds (71.4%), the exception is thrown. Similar to the array allocation, no pattern entirely preventing the issue could be found, and the original AST were mined for positive patterns. For read, four patterns were identified in Figure 6.17. One with a variable read for a single dimensional array (read a), a two-dimensional array (read b), a division of a read by a literal (read c) and an addition or subtraction of a variable with another variable

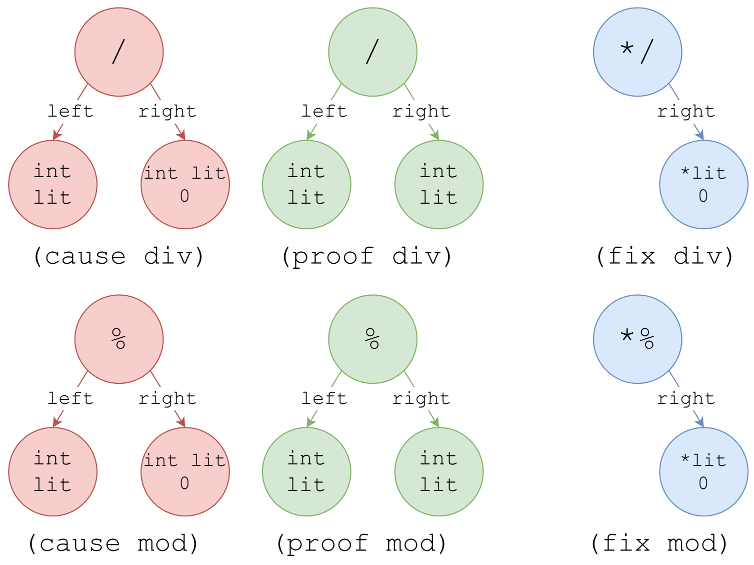


Figure 6.18: An Arithmetic exception happens when a division by 0 (top left) or a modulo by 0 (bottom left) is conducted. The fix is to omit the 0 literal in the division (top right) and modulo (bottom right) of all data types. To prove the anti-pattern during the verification phase a positive pattern is applied as well, forcing the AST generation to consist of literals on both sides of the operations (middle top and bottom).

or literal (read d) (confidence 79.8%). For write, four similar patterns were identified, with the first three identical to read (write a-c), and the fourth being slightly restricted as only a literal was added or subtracted from the variable (write d) (confidence 82.7%). Similar to the allocation issue, this does not prevent the bug, as out of bounds exceptions can still happen with negative variables, as well as values that are too high. The patterns in this section only tackle the symptom but not the root cause of the issue. It is questionable if the locations where the variables are being set outside the array range can be identified with static analysis at all.

Arithmetic Bug Patterns

The *Arithmetic* exception happens only in 9 AST, and is caused by two patterns, as shown in Figure 6.18. The first cause is a division by 0 (98.6% confidence), and the second is a modulo by 0 (98.5% confidence). Both patterns were only identified with int, and modulo division. The corresponding fix was implemented data-type-independently for each type preventing the corresponding 0 value, e.g., 0 for int, and 0.0 for float and double. To ensure that this anti-pattern works correctly, the fix-verification was conducted with patterns in addition to the anti-pattern to be prevented. The patterns enforce a division (100% confidence) or modulo (99.3% confidence) with int literals on both sides, to prevent other bugs from occurring during verification. In the one instance where modulo failed, the AST was generated as a divisor of a division, with the modulo resulting in 0, showing that even in a rather simple anti-pattern producing a 100% confidence is not guaranteed.

Unprovable Bug Classes

For *Timeout*, *Process crash with log*, and *Process crash without log* all identified patterns turned out to be false positives during pattern verification.

For the *Timeout* class, primarily anti-patterns were identified that concerned simple math expressions such as addition or subtraction, or allocation and access to simple variables and arrays. None of them provably affected a timeout. While they may have some foundation in leading to endless loops, the connection to loops was not identifiable via patterns. Curiously, some patterns identified as fault of omission contained loops, identifying loops as something that will not lead to a timeout. There may be two reasons why the timeout was not verifiable. The first is that the difference between successful ASTs and those running into a *Timeout* was not large enough for a mining approach depending on differences between the ASTs. The second may be that run time as a NFP is so specific that a mining can only happen under different circumstances, such as only considering one experiment group, or single algorithm instead of the entire suite of experiments. To mitigate the first possible issue, the timeout for future experiments will be increased. The second issue will be addressed in Section 6.5.

For *Process crash with log* as well as *Process crash without log*, all identified anti-patterns were not provable. The only indicator that no issue in the ASTs was responsible, was that for every attempt at pattern validation the *process crash with log* occurred on average in 1% of all runs. After investigation, this turned out to be an issue in the transmission between the separate JVMs for evaluation in the Amaru framework, and was fixed. As *Process crash without log* has never occurred in following experiments and the occurrence of *Process crash with log* is significantly reduced, it is assumed that the issue was causing both bug classes.

Summary

Out of 12 defined classes of mutatoinal bugs, for at least 9 of them patterns and corresponding solutions for them were successfully identified. Table 6.10 summarizes the identified patterns shortly, and gives the identified confidence for it. During validation of the reason the issues occured, the worst confidence was 71.9%, and the best was 98.6%. In most cases, only one single issue was identified, and it can be assumed that less often occurring issues were obscured by more often occurring ones.

Applied fixes were validated with a confidence between 82.7% and 100%. A confidence of 100% is not always guaranteed due to the complexity of software. Much of the validation was impacted by co-occurring bugs, such as Access out of bounds being impacted by arrays being allocated at all, making it challenging to isolate the effects of patterns. Branches and dead code are also a challenge (see Figure 6.10).

The amount of ASTs that are available for a mining seem to be irrelevant. Between 3,380 ASTs of the largest group and only 9 AST in the smallest group, led to the identification of patterns. Most patterns in the larger groups were co-located with similar patterns, which occurred less in the smaller groups.

These results show that pattern mining can be successfully identified directly at a native representation of a compiler or interpreter, and can be conclusively proven or disproven. The presented basis has room for improvement, such as switching to a dynamic approach, also analyzing which nodes are activated and how often, as well as including this information in the pattern verification itself.

Table 6.10: Confidence scores for all identified bug patterns, and corresponding patterns preventing them.

Exception	Pattern	Conf.	Resolution	Conf.
Illegal State	Read without preceding write	82.7%	Generate read only with preceding write available	94.3%
Illegal Argument	Read of stack variable on heap	97.4%	Validation which node can access which variable	100%
Process crash with log	Issue in Framework not AST	-	Framework fix	-
Unsupported Specialization	function called with incorrect arguments	96.7%	Correct arguments extracted from \gls{acr:ast}	99.3%
Timeout	no pattern identified	-	-	-
Class Cast	array read array of wrong type, char or string	93.5%	Generate read only when corresponding type write available	85%
	char read of string or array	94.1%	char excluded in experiments	-
	string read of char or array	95.8%	string excluded in experiments	-
	cast of char[] string casting wrong type	97.1%	cast excluded in experiments	-
	cast of any to float	97.4%	cast restricted to cast from float to float only	90.2%
Exception not serializable	Array allocated with negative size	97.3%	Patterns to allocate array more sensibly	93%
Null Pointer	char read	77%	char excluded in experiments	-
	string read	77.7%	string excluded in experiments	-
	array read without preceding write	85.9%	Generate array read only with preceding array write available	97.9%
Array Index Out Of Bounds	Array read out of bounds	79.8%	Patterns to read array more sensibly	86.6%
	Array write out of bounds	71.4%	Patterns to write array more sensibly	82.7%
Process crash without log	Issue in Framework not AST	-	Framework fix	-
Runtime	non-existing function called	91%	functions identified from function registry	100%
Arithmetic	division by zero	98.6%	0 literal excluded for divisor	100%
	modulo by zero	98.5%	0 literal excluded for modulus	99.3%

6.4 Application of Patterns in KGGI

The next experiment in the series aims to answer the research question how identified patterns can be utilized to lead to general optimizations. In this case, the bug patterns from the previous sections are applied to see how they influence the search space in KGGI to improve the overall experiments, as well as to see if these experiments then yield semantically correct ASTs with an improved run time performance. To enable comparability to the baseline, the exact same algorithm suite of 25 experiments is used. The same parameters as outlined in Section 6.2 are applied. The following differences apply:

Patterns All three major operators have the *anti-patterns* as well as *patterns* identified injected. This includes the *string* and *char* data type anti-patterns, which are injected as anti-patterns rather than restricting the search space outright. This was done purely for comparability to the baseline. Due to how KGGI works, anti-patterns can still occur if the algorithm has no choice other than limiting the amount of generated anti-patterns. If the search space were to be restricted, the generation of *string* or *char* data types would be impossible during the experiments. A total of 9 anti-patterns and 11 patterns are applied.

Crossover The crossover is left as a single-point crossover. Unlike the baseline crossover, however, the crossover now adheres to patterns. This is done via Algorithm 11 as outlined in Section 5.5. The crossover uses the algorithm to identify anti-pattern locations via a simulated crossover of the selected ASTs. If the simulated child-AST has an equal or lower amount of anti-patterns than identified in the left parent, e.g., the parent that has the single-point injected from the right parent, then the crossover will actually be conducted. If after 5 attempts no such simulation is successful, the crossover instead attempts to select only crossover points without anti-patterns in them for another 5 attempts before failing entirely.

Mutator The crossover only considers anti-patterns. The mutator considers both types of patterns. While anti-patterns are always active and prevented as far as possible (for example existing anti-patterns are not attempted to be mutated away), patterns instead have an activation chance of 33%. Since KGGI works recursively, a pattern has multiple chances to be activated during sub-AST generation. If multiple patterns are competing, for example the four identified read anti-patterns, only one can be activated instead of multiple ones at the same time. However, the mutator still randomly selects a mutation point, which means that in some selections patterns may not be applicable at all, or that one or more anti-patters are already in the AST or may be unavoidable. KGGI uses the pattern identification algorithm in the mutator to provide this information at the mutation point. The newly generated sub-AST is then generated with a minimum viable number of anti-patterns, in most cases none.

Fitness Function The fitness function is the accuracy on the test cases as defined in Algorithm 1, exactly as in the baseline. The reason why performance is not part of the fitness function, even though this work analyzes NFP, is to find multiple variants of the algorithms being tested in the experiment suites that have a diverse run-time behavior. The pattern mining approach requires a margin between the groups to work, as evidenced by finding no valid pattern for *Timeout*.

Timeout To improve the chance of finding patterns related to run-time performance, the timeout was increased from the values used in the baseline experiment. For the *math* and *sort* experiment suites, the timeout was set to 10 seconds (previously 3). For the *nn* experiment suite, the timeout was set to 30 seconds (previously 10).

Over the experiments utilizing KGGI with patterns, 37,194 different solutions were created, which is an improvement of 21,288 over the baseline. Of these solutions 10,845 succeed every test case, 13,725 produce a run-time exception and 12,624 produce invalid results during at least one test. This means that the diversity, as well as the amount of successful solutions is more than doubled when applying patterns. In total, all ASTs consist of 9,305,089 nodes. Even though the amount of distinct ASTs more than doubles, this is 1,064,053 nodes less than the baseline. This is another indicator (in addition to the results of the pattern verification in the previous section) that the baseline ASTs have many branches that are never executed.

Table 6.11: Operation success rates in KGGI with patterns. The percentages show the current percent of operations on the left, compared to the original percentage from Table 6.4 on the right. For the successful and failed test groups an ↑ shows an improvement, and for run-time exception group the ↓ shows an improvement.

	Successful	**Failed test**	**Run-time exception**	**Total**
Create	434 (17,2% ↑ 12,1%)	792 (31,4%↑ 26,8%)	1,299 (51,4% ↓ 38,9%)	2,525
Mutate	2,168 (35,5% ↑ 22%)	1,433 (23,5% ↑ 18,7%)	2,508 (41% ↓ 40,7%)	6,109
Crossover	19,756 (48% ↓ 16,8%)	10,756 (26,2% ↑ 11,6%)	10,603 (25,8% ↑ 5,2%)	41,115

Table 6.11 shows the results of the individual operations in the experiment. Both the create and mutate operation are vastly improved, producing more successful variants, and fewer variants with run-time exceptions. The crossover becomes worse, with 16,8% fewer operations producing successful results. While this means that still nearly every second crossover produces a successful variant, now one quarter fails at least one test case and another quarter produces run-time exceptions. Since the crossover is the most applied operation, this is a drastic decrease in quality. It is likely that since the diversity of options in the suite is now much higher, the crossover has a higher chance of crossing mutants that are more different from the original and thus have more chances of failing than in the baseline. Figure 6.19 shows the histogram of all unique solutions per generation in the KGGI experiment utilizing patterns on shaker sort. If compared with Figure 6.8 the amount of successful solutions is drastically improved, and every generation now has a much higher diversity rate instead of only the first generation. This behavior is similar in all 25 experiment runs, with some differences to the baseline:

- The *math* experiment suite tends to have the most successful solutions overall, but the least in the first four generations. Similar to the baseline, some individuals also return the wrong data type.
- In the *sort* experiment suite, the first generation tends to have more successful solutions than the other two suites. Other than the

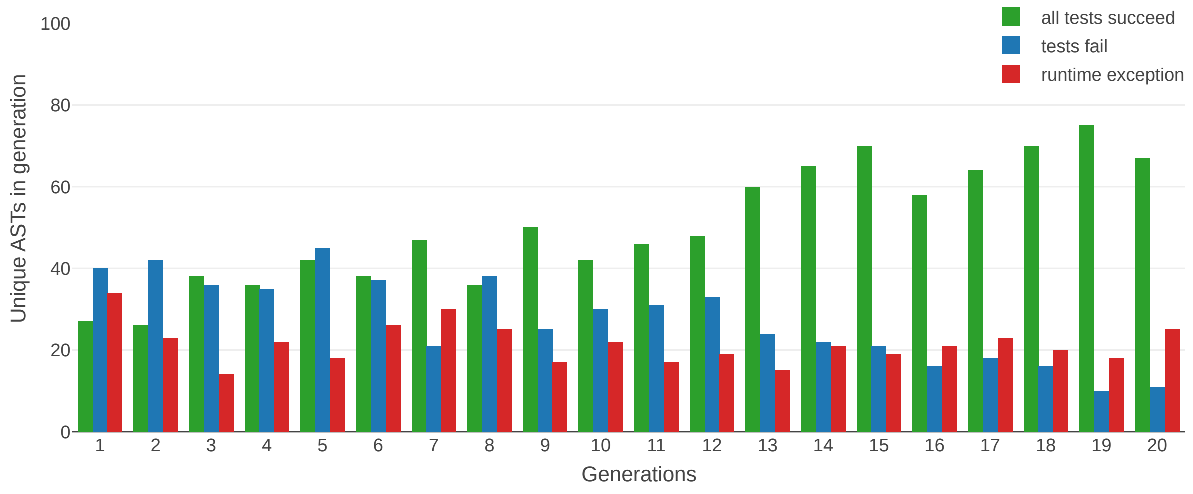


Figure 6.19: Histogram of unique solutions per generation in the KGGI experiment run of *shaker sort using patterns* for comparison with Figure 6.8. Uniqueness is considered in the generation, not overall. The groups are *individuals succeeding all tests*, *failing at least one test*, or producing a *run-time exception* in at least one test.

Table 6.12: Unique AST individuals in math experiment using patterns in KGGI. The second line shows the difference to the baseline. ↑ is an improvement in successful, failed test and total. ↓ is an improvement in run-time exception.

Experiment	Successful	Failed test	Run-time Exception	Total
Square root - Java	551(58,7%) ↑273 (↑11,4%)	113 (12,1%) ↑90 (↑8,1%)	274 (29,2%) ↓12 (↓19,5%)	938 ↑ 351
Square root - lookup table	184 (12,6%) ↑20 (↓14,2%)	503 (34,4%) ↑458 (↑27%)	776 (53%) ↑373 (↓12,8%)	1,463 ↑ 851
Square root - regular	792 (47,5%) ↑418 (↑3,2%)	389 (23,3%) ↑342 (↑17,8%)	486 (29,2%) ↑63 (↓21%)	1,667 ↑ 823
Cube root	525(30,6%) ↑178 (↓13,9%)	579 (33,7%) ↑481 (↑21,2%)	613 (35,7%) ↑277 (↓7,3%)	1,717 ↑ 936
Super Root	565 (34,3%) ↑564 (↑34,1%)	490 (29,7%) ↑285 (↓6,2%)	594 (36%) ↑229 (↓27,9%)	1,649 ↑ 1,078
Inverse Square Root	759 (45%) ↑521 (↑14,1%)	443 (26,3%) ↑338 (↑12,7%)	484 (28,7%) ↑56 (↓26,8%)	1,686 ↑ 915
Logarithm 10	659 (38,2%) ↑421 (↑4,1%)	298 (17,3%) ↑283 (↑15,1%)	768 (44,5%) ↑323 (↓19,2%)	1,725 ↑ 1,027
Logarithm Naturalis	906 (52,4%) ↑756 (↑25,4%)	229 (13,2%) ↑211 (↑10%)	596(34,4%) ↑208 (↓35,4%)	1,731 ↑ 1,175
Total	**4,941 (39,3%)** **↑3,151 (↑6,3%)**	**3,044 (24,2%)** **↑2,488 (↑13,9%)**	**4,591 (36,5%)** **↑1,517 (↓20,2%)**	**12,576** **↑7,156**

increased success rate, the behavior is similar to the baseline.

- While improved, compared to the baseline, the *neural network* experiment suite still has the lowest diversity of all three suites. Unlike the baseline, where the amount of successful individuals tended to decrease over the generations, the amount of successful solutions now seems to be more constant.

In the math experiment suite, as shown in Table 6.12, 12,576 out of approximately 16,000 expected unique AST are generated, more than double the baseline. While all individual experiments show more diversity, *square root - java* shows the least improvement, with only about 30% more individuals. This might be due to the search space being too restricted, as the depth and width restrictions stem from the original AST size. Overall, approximately two out of five individuals are successful, an improvement over the baseline of 6,3%. Due to the higher diversity than the baseline, this means that more than twice as many successful individuals are generated. Although *square root - lookup table* and the *cube root* are now less successful in percentage compared to the baseline, all experiments yield more successful individuals than the baseline. *Super root* shows the most improvement, jumping from just one successful individual in the baseline to 565 individuals. However, this may be due to an outlier in the baseline. The amount of run-time exceptions also drops by around one fifth compared to the baseline. In this experiment set, utilizing patterns in KGGI is an overall success.

The sort experiment suite (see Table 6.13), shows a similar overall structure as the math experiments. With 16,221 out of an expected 20,000 AST it has the highest diversity of all experiment suites. *Heap sort* is still the worst performing experiment in percentage of successful solutions. *Bubble sort* and *merge sort inlined* now perform much worse than the baseline, producing less successful variants than the baseline overall. All other sorting algorithms now show more successful variants, with *heap-*, *bubble-*, *merge-* and *selection sort* showing a decrease in percentage.

Table 6.13: Unique AST individuals in sort experiment using patterns in KGGI. The second line shows the difference to the baseline. ↑ is an improvement in successful, failed test and total. ↓ is an improvement in run-time exception.

Experiment	Successful	Failed test	Run-time Exception	Total
Bubble	203 (13,4%) ↓60 (↓23,4%)	808 (53,5%) ↑734 (↑43,1%)	499 (33%) ↑123 (↓19,7%)	1,510 ↑797
Heap	216 (11,9%) ↑124 (↓3,1%)	1,043 (57,4%) ↑897 (↑33,6%)	558 (30,7%) ↑182 (↓30,5%)	1,817 ↑951
Insertion	397 (25,4%) ↑272 (↑3,7%)	681 (43,5%) ↑623 (↑33,5%)	487 (31,1%) ↑93 (↓37,2%)	1,565 ↑988
Merge	622 (35,3%) ↑405 (↑2,2%)	650 (36,9%) ↑570 (↑24,7%)	491 (27,8%) ↑132 (↓26,9%)	1,763 ↑1,107
Merge Inlined	226 (15,4%) ↓284 (↓39,9%)	786 (53,7%) ↑697 (↑44%)	452 (30,9%) ↑129 (↓4,2%)	1,464 ↑542
Quick	755 (42,4%) ↑582 (↑15,4%)	230 (12,9%) ↑216 (↑10,7%)	797 (44,7%) ↑343 (↓26,1%)	1,782 ↑1,141
Quick Inlined	350 (23,7%) ↑237 (↑2,1%)	261 (17,7%) ↑242 (↑14%)	866 (58,6%) ↑476 (↓16,1%)	1,477 ↑955
Selection	491 (31,1%) ↑183 (↓10,4%)	609 (38,6%) ↑524 (↑27,2%)	477 (30,3%) ↑128 (↓16,8%)	1,577 ↑835
Shaker	465 (32,4%) ↑313 (↑8,9%)	546 (38,1%) ↑463 (↑25,2%)	423 (29,5%) ↑13 (↓34,1%)	1,434 ↑789
Shell	952 (52%) ↑671 (↑14%)	372 (20,3%) ↑310 (↑11,9%)	508 (27,7%) ↑110 (↓26%)	1,832 ↑1,091
Total	**4,677 (28,8%)** **↑2,443 (↓4,2%)**	**5,986 (36,9%)** **↑5,276 (↑26,4%)**	**5,558 (34,3%)** **↑1,729 (↓22,2%)**	**16,221** **↑9,488**

Overall, the amount of successful solutions is more than double the baseline, and the amount of ASTs is reduced to about one third of all AST in the populations. While the percentage of successful solutions is slightly decreased, the doubled amount of successful ASTs now generated sill shows a successful improvement over the baseline when applying patterns in KGGI.

The *neural network* experiment suite is overall the least successful of the three suites, while simultaneously having the greatest difference to the baseline. As can be seen in Table 6.14, similar to the other experiments the diversity of the populations is more than doubled, and also the amount of successful solutions nearly triples. With only a 14,6% rate of successful individuals in the population and a 42,6% exception rate overall the suite is less successful than the other two suites. What is notable, is that the experiment with the *sigmoid* activation function now has 45 successful individuals, resulting in all experiments now being successful. *Tanh* has the largest improvement overall with an increase of 14,1% in successful solutions, and a 42,1% decrease in run-time exceptions.

Exceptions in KGGI With Mutational Bug Patterns

The experiment results show that applying patterns in KGGI can improve both the amount of valid results, and the diversity of the population, while simultaneously decreasing the occurrence of run-time exceptions. The question remains if the individual patterns and anti-patterns that were identified and applied, fulfilled their purpose. As KGGI in both the crossover and mutator still might introduce anti-patterns, it is not expected that all identified exception classes will not be present anymore.

Table 6.14: Unique AST individuals in neural network experiment using patterns in KGGI. The second line shows the difference to the baseline. ↑ is an improvement in successful, failed test and total. ↓ is an improvement in run-time exception.

Experiment	**Successful**	**Failed test**	**Run-time Exception**	**Total**
Rectified Linear Activation	120 (9,6%) ↑55 (↓2,6%)	597 (48%) ↑491 (↑27,9%)	528 (42,4%) ↑170 (↓25,3%)	1,245 ↑716
Leaky Rectified Linear Activation	194 (16%) ↑100 (↓1%)	482 (39,8%) ↑408 (↑26,4%)	536 (44,2%) ↑151 (↓25,4%)	1,212 ↑659
Sigmoid	45 (3,7%) ↑45 (↑3,7%)	599 (49,2%) ↑463 (↑22,2%)	573(47,1%) ↑205 (↓25,9%)	1,217 ↑713
Swish	219 (19,2%) ↑123 (↑0,1%)	396 (34,6%) ↑334 (↑22,3%)	529 (46,2%) ↑182 (↓22,4%)	1,144 ↑639
Tanh	183 (15,5%) ↑175 (↑14,1%)	676 (57,4%) ↑514 (↑28%)	319 (27,1%) ↓63 (↓42,1%)	1,178 ↑626
Fully Inlined NN	262 (21,7%) ↑155 (↑2,3%)	424 (35,2%) ↑349 (↑21,6%)	519 (43,1%) ↑150 (↓23,9%)	1,205 ↑654
All Activation Functions	204 (17,1%) ↑97 (↓2,4%)	420 (35,1%) ↑345 (↑21,5%)	572 (47,8%) ↑203 (↓19,1%)	1,196 ↑645
Total	**1,227 (14,6%)** **↑757 (↑2%)**	**3,594 (42,8%)** **↑2,918 (↑24,6%)**	**3,576 (42,6%)** **↑1,009 (↓26,6%)**	**8,397** **↑4,684**

Smaller anti-patterns also may not have been found in the bug mining experiment, since the effects of larger issues may have overshadowed them.

Table 6.15 and Figure 6.20 show the new distribution of exceptions in the experiment suite when patterns are applied. In the table, the percentages in the brackets relate to the amount of ASTs in the entire population, not just the failing AST. The largest exception class *timeout* drastically increased compared to the baseline. As no pattern was identified causing this exception, it was also never prevented by KGGI. Three new exceptions occur now, which are JVM errors. *Assertion*, *Stack Overflow* and *Out of Memory* would have been identified as *Exception not serializable* in the baseline. This also means that during analysis of that class, at least the *Stack Overflow* and *Out of Memory* exceptions were not identified as separate patterns from the *Assertion*. Since the transmission error has

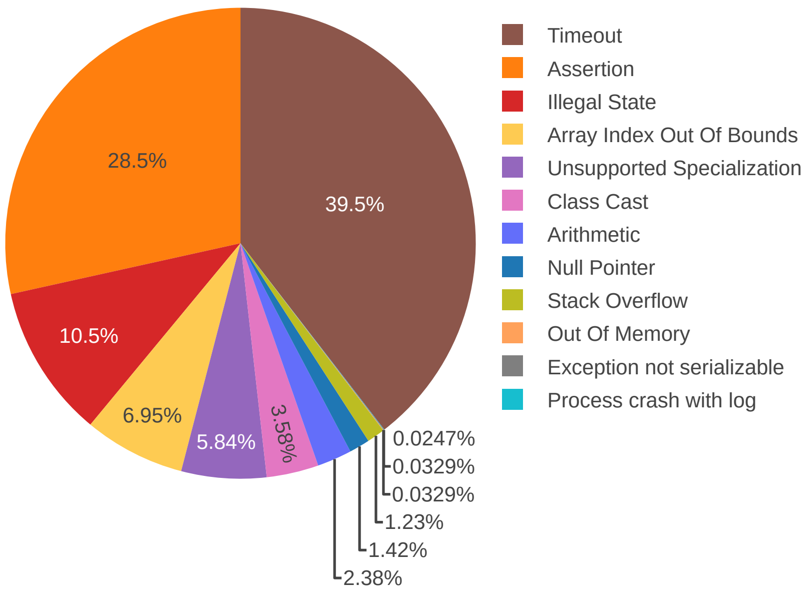


Figure 6.20: Distribution of the exception groups in percent according to the occurrences in Table 6.15

Table 6.15: Overview of the number of **bugs** that occur over all experiments with KGGI when utilizing patterns compared to the baseline. In the difference, ↓ is better. Occurrences are counted per unique AST. Different test cases may return different exceptions, thus ASTs may occur in multiple groups.

Exception	ASTs	Baseline	Difference
Timeout	4,805 (12.9%)	481 (3.0%)	↑4,324 (↑9.9%)
Assertion E	3,460 (9.3%)	-	↑3,460 (↑9.3%)
Illegal State	1,279 (3.4%)	3,380 (21.2%)	↓2,101 (↓17.8%)
Array Index Out Of Bounds	845 (2.3%)	92 (0.6%)	↑753 (↑1.7%)
Unsupported Specialization	710 (1.9%)	513 (3.2%)	↑197 (↓1.3%)
Class Cast	435 (1.2%)	398 (2.5%)	↑37 (↓1.3%)
Arithmetic	289 (0.8%)	9 (0.1%)	↑280 (↑0.7%)
Null Pointer	172 (0.5%)	225 (1.4%)	↓53 (↓1.0%)
Stack Overflow E	149 (0.4%)	-	↑149 (↑0.4%)
Out of Memory E	4 (0.0%)	-	↑4 (↑0.0%)
Exception not serializable	4 (0.0%)	368 (2.3%)	↓364 (↓2.3%)
Process crash with log	3 (0.0%)	867 (5.5%)	↓864 (↓5.4%)
Illegal Argument	-	3,208 (20.2%)	↓3,208 (↓20.2%)
Process Crash without log	-	30 (0.2%)	↓30 (↓0.2%)
Runtime	-	19 (0.1%)	↓19 (↓0.1%)
Total	12,155 (32.7%)	9,590 (60.3%)	2,565 (-27.6%)

been fixed, the three classes occur now, but similar to *Timeout* had no patterns preventing them.

The third-largest group, *Illegal State*, originally the largest, occurs in only 3.4% of the population instead of 21.2% in the baseline. While the identified anti-patterns are mostly prevented, not all of them are. This is still a significant improvement. The originally second-largest group of exceptions, *Illegal Argument* was completely prevented via patterns and does not occur anymore at all. It seems that all reasons for this bug were identified correctly, and KGGI now prevents them. The same goes for *Runtime* which was successfully prevented, although this was originally the second-smallest exception class.

Process crash without log is now also prevented completely. However, this has nothing to do with patterns, but rather with applying fixes to the communication code between the experiment runners of the Amaru framework. This is the same reason why *Process crash without log* now occurs with 5.4% less frequency. Together with the *Exception not serializable* class, which still rarely occurs, these three classes were fixed manually as they had nothing to do with the actual ASTs being tested or with applied patterns. The group *Exception not serializable* was only partially caused by a transmission error, as there were real bugs also causing the same exception type.

The results for the remaining exception classes are inconclusive concerning the application of patterns. *Index out of Bounds* as well as *Arithmetic* are the only two exception types where patterns were identified and applied, yet the exception now occurs more frequently than before. In the case of *Index out of Bounds* no anti-patterns were prevented, but rather positive patterns were applied. In the bug mining experiment it was already asserted that the positive patterns only tackle a symptom, e.g., how the read or write for the array is being generated, rather than the actual problem, which was that the variable being read was previously assigned a value outside the range of the array. In the case of *Arithmetic*

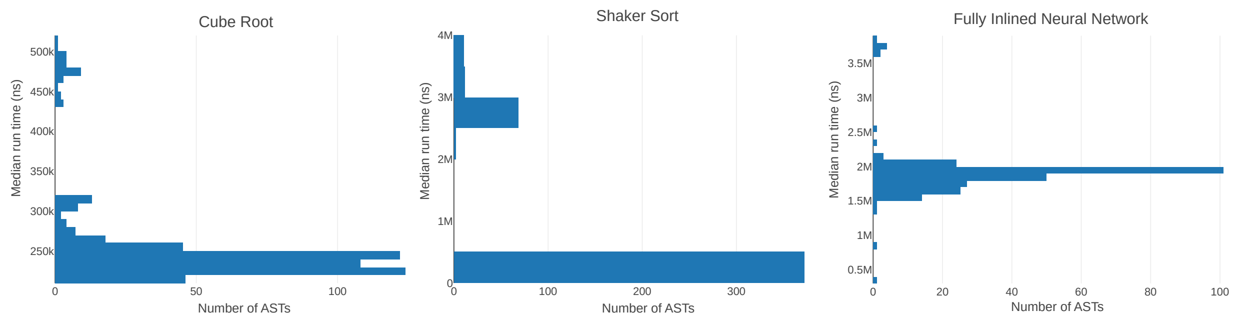


Figure 6.21: Median run-time performance distributions of all ASTs in one experiment. Shown are representatives per group. *Cube root* for *math* (left), *Shaker Sort* for *sort* (middle), and *Fully Inlined* for *neural networks* (right).

this is likely the same reason why this bug now occurs more often. While the division and mod by zero exceptions are prevented via patterns, ASTs were generated that either read 0 from a variable, or have more convoluted sub-AST that produce a 0 as the divisor.

The final exception classes all now occur approximately half as often concerning the percentage of the population, which means that the applied patterns are still successful. *Unsupported Specialization* might still occur, since the array data types can't be successfully identified during function calls. *Class Cast* may still occur for the same reason. The MiniC node CopyArray is supposed to copy any array from the parameters of a function call to the stack. However, since it is rather generic, it may also just copy an array into a variable intended for a different data type or array type. Finally, *Null Pointer* may still occur for the same reason *Illegal State* still occurs. It is caused by reading an uninitialized array, which still can happen even though the likelihood was decreased.

Run-Time Performance

All successful ASTs are profiled 200,000 times. This includes those succeeding at every test case for *math* or *sort*, as well as those reaching sufficient quality for *neural network* within a 1% margin. The following information is presented, with the first 100,000 runs discarded and only the second 100,000 used in the data. To restrict the benchmark time for very unsuccessful ASTs the timeout was set to 5 minutes for the *math* experiments and to 30 minutes for the *sort* and *neural network* experiments. The best ASTs identified in the experiments are listed in Appendix A.

Over all runs a rather similar behavior for each KGGI experiment is shown, at least for each individual experiment group. Figure 6.21 shows one representative run-time distribution for each experiment group, presented in bins. For all three, the majority of solutions has a run time similar to the original.

In the *math* functions, there are additional outliers that have a far higher run-time performance (around 20-30 ASTs). For *square root java*, this is only one individual AST (448,000 nanoseconds vs 3,000 nanoseconds). *Inverse square root* has an additional outlier group of 22 ASTs that significantly outperform the baseline. *Logarithm naturalis* and *Logarithm 10* have the lowest range of run-time performances of only 60,000 nanoseconds and 71,000 nanoseconds respectively. *Super root* has 442 ASTs that had a timeout during profiling, meaning that the majority of generated ASTs was significantly worse than the original (> 1.5 million nanoseconds).

The *sort* functions show a similar behavior, although the outlier group with a worse performance is larger (100 ASTs). *Quick sort* hand 624 timeouts during profiling. *Quick sort inlined* had 280. In both cases almost all of the AST did produce a timeout.

For all *neural network* experiments, the distribution deviates from *math* and *sort*. One additional group is far more successful concerning run-time, for every original individual in the experiment. However, both outlier groups are relatively small, with around 2 to 3 ASTs each.

The identified distributions should be applicable for a differential mining approach, as a sufficient amount of ASTs can be grouped between *efficient* and *inefficient* concerning their run-time performance.

Most of the experiments achieve an improvement of the run-time performance compared to the baseline. Table 6.16 gives an overview of the peak performance (minimal measurement over 100,000 measurements) as well as the median performance, which can be considered to be the regular behavior of the AST. All the performances were compared with the baseline via a Mann-Whitney-U test for parametric, independent values, which all the run-time measurements are. Over the entire 100,000 executions, all the values have a p score of 0.0. To ensure that there is no overfit, the tests were also conducted with 10,000, 1,000 and 100 randomly selected samples of the entire 100,000 measurements. The only instance where a p-value of more than 0.001 is achieved, is *Square root - lookup table* with a value of 0.054 with 100, and a p-value of 0.185 with 1,000 samples. This means that except for this one instance, all performance improvements are statistically significant.

In the *math* group, only *Square-root java*, and *Square root lookup table* are not improved. With *Square root lookup table* there is an improvement in the peak performance, but this can just be an outlier, as the p-value in this group is the only invalid one. *Inverse Square Root* shows the largest improvement of the group, with a 60% improvement. The function is twice as performant, while having the same accuracy as the original function. In all other functions, there is a minimal improvement of around *5%*. The performance, compared to the baseline, is shown in Figure 6.22.

The *sort* algorithms already have a significant skew in how their performance is, from the worst with emphBubble sort being more than 1,000x worse than *Shell sort*. All the algorithms that have a high run-time are only slightly improved. The already highly performant algorithms however improve their run-time by around *53%* for *shaker sort* and by around *82%* for *shell Sort*, as shown in Figure 6.23. *Quick sort inlined*, as well as *selection sort* do not manage to outperform the peak performance, but have a significantly improved median performance. *Merge sort* is the only sort algorithm that is not improved at all.

The *neural network* algorithms show the highest runtime improvement overall in Figure 6.24. The worst improvement is *Sigmoid* with only *15%*, and most other variants show an improvement between *50%* and *80%*. The greatest improvement overall is *Swish* with *99.83%* improvement over the original runtime. This means the algorithm would be 592x faster

Table 6.16: Overview of the peak and median performances of the baseline AST and the best counterpart generated via KGGI. Percentages are given compared to the baseline. (-) means an improvement, (+) means that the best found individual is worse than the baseline. All values are given in nanoseconds.

Function	Original peak perf,	Best found peak perf,	Peak %	Original median perf,	Best found median perf,	Median %
Square root - java	2,524	2,614	4%	2,553	2,647	4%
Square root - lookup table	25,348	24,957	-2%	26,449	28,804	9%
Square root - regular	146,175	138,267	-5%	147,868	139,029	-6%
Cube root	228,447	216,090	-5%	231,039	217,273	-6%
Super Root	314,578	298,909	-5%	316,342	300,622	-5%
Inverse Square Root	191,028	75,480	-60%	192,140	75,920	-60%
Logarithm	274,815	269,106	-2%	282,900	269,938	-5%
Logarithm Natural	513,065	510,345	-1%	549,841	510,745	-7%
Bubble	2,895,883	2,693,669	-7%	3,492,836	2,789,038	-20%
Heap	277,765	266,676	-4%	304,806	279,540	-8%
Insertion	2,779,711	2,664,799	-4%	3,382,195	2,760,911	-18%
Merge	298,180	409,756	37%	376,408	437,187	16%
Merge inlined	76,644	56,232	-27%	83,477	60,620	-27%
Quick	1,179,138	1,125,290	-5%	1,297,887	1,174,222	-10%
Quick inlined	1,090,232	1,114,813	2%	1,247,939	1,161,199	-7%
Selection	2,287,075	2,304,079	1%	2,772,077	2,355,165	-15%
Shaker	10,019	4,438	-56%	11,762	5,580	-53%
Shell	34,104	5,881	-83%	36,909	6,672	-82%
Rectified Linear Activation	1,434,476	747,061	-48%	1,574,603	806,403	-49%
Leaky Rectified Linear Activation	1,444,317	367,122	-75%	1,564,061	394,223	-75%
Sigmoid	1,472,688	1,249,746	-15%	1,565,001	1,315,781	-16%
Swish	1,811,362	3,055	-100%	1,924,867	3,166	-100%
Tanh	1,586,832	864,352	-46%	1,683,544	991,951	-41%
Fully Inlined NN	1,920,321	381,592	-80%	1,982,779	395,537	-80%
NN with all Activation Functions	1,650,032	407,326	-75%	1,718,321	429,988	-75%

than the original, which is unrealistic. While the other algorithms may be explainable due to overfit, *Swish* abuses an exploit. Since the training data is passed to the function under optimization, this particular AST simply returns the training-output instead of conducting the training. This can be considered a severe overfit, which is a known problem in the GI domain [110]. A detailed analysis follows in the next section.

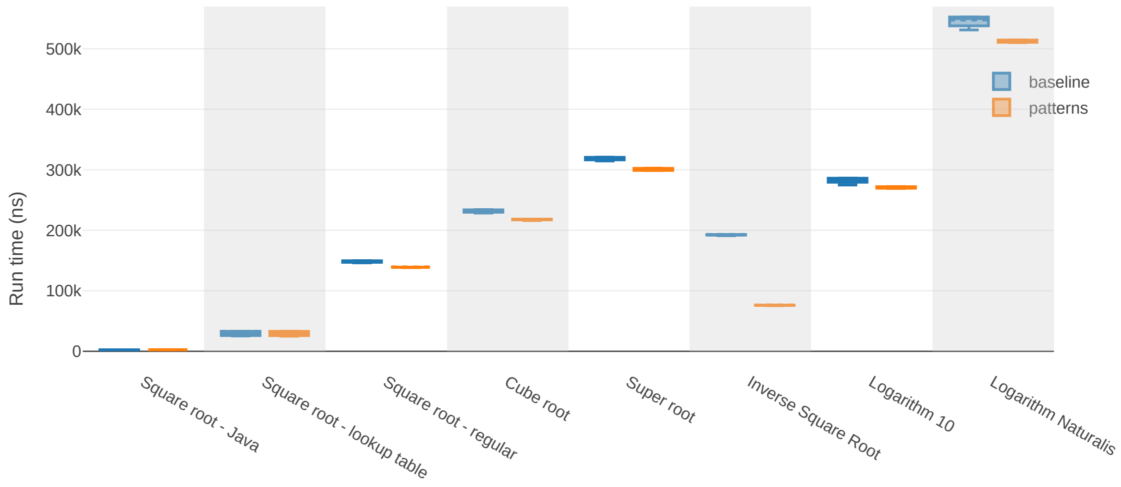


Figure 6.22: Performance of the synthesized *math* ASTs (orange) compared to the originals (blue).

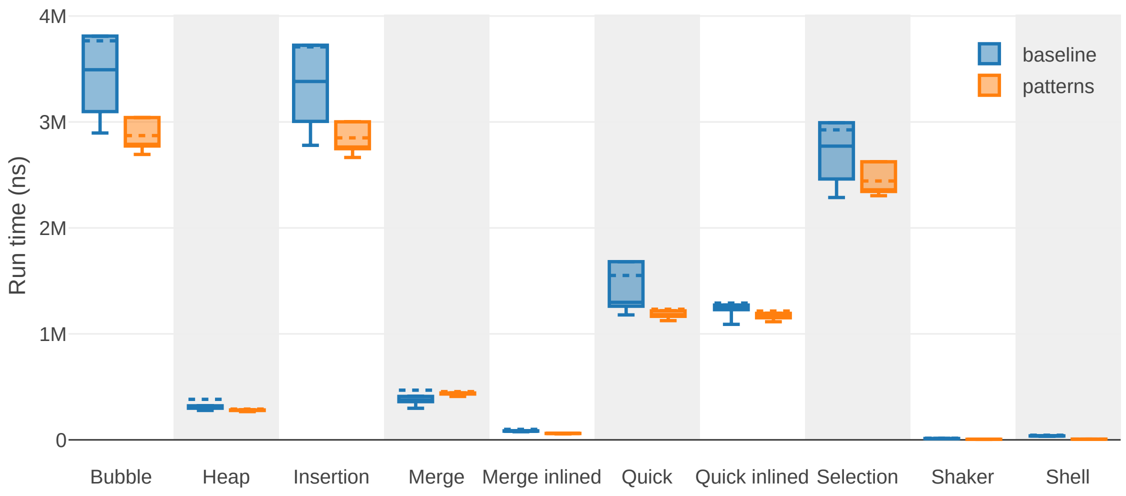


Figure 6.23: Performance of the synthesized *sort* ASTs (orange) compared to the originals (blue).

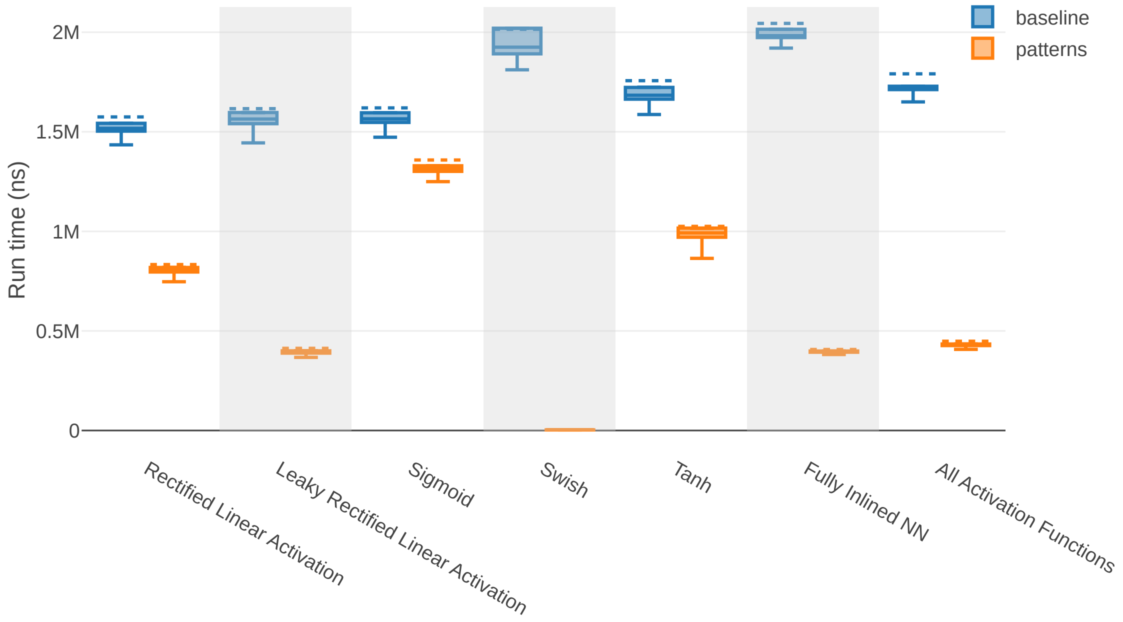


Figure 6.24: Performance of the synthesized *neural network* ASTs (orange) compared to the originals (blue).

6.5 Mining of Performance Patterns

The final experiment in the series attempts to identify recurring patterns in the domain of NFP - *performance*. The goal is to identify recurring patterns that are either responsible for increasing the run time, e.g, *inefficient* anti-patterns, or patterns that decrease the run time, e.g., *efficient* patterns. Only such AST are mined, that are valid, meaning that they succeed at every test case for *math* or *sort*, or are within a 1% margin for *neural network*.

To ensure that the mining can be discriminative, every mining is conducted with three different groups, depending on the identified execution time over all ASTs over each group. This is done by grouping the executed ASTs into quartiles. From the quartiles, the interquartile range (IQR) is calculated (distance between 3rd and 1st quartile):

Efficient ASTs are those whose median execution time is less than the median - 2 IQR. For all groups where this did not leave any efficient algorithm, the lower limit was raised manually by reviewing the ranges. This affected *rectified linear activation*, all *sort* and all *math* algorithms except *Inverse Square Root*.

Inefficient ASTs are those whose median execution time is more than the median + 2 IQR. For some algorithms, this had to be changed. *Heap sort*, *Logarithm Naturalis*, and *Super Root* had the limit lowered as there was no AST outside the range. For *Square root - Java and lookup table*, *Logarithm 10*, and all remaining sort operations the limit had to be raised, since the amount of ASTs was too high, obscuring any pattern in the *efficient* groups.

Timeout ASTs that were so inefficient that their run-time profiling timed out were added to their own group.

The experiment is conducted using the IGOR algorithm. Unlike the previous bug-mining experiment, the search space is much more manageable, as only the succeeding ASTs which are discriminative in their run time will be mined.

The experiment is conducted in three layers. For *group* and *total* the ASTs are just combined from the individual analysis as described above. Considering median and IQR would not make sense due to the large differences in run time. Also, for those two categories the mining was conducted twice, once with *inefficient* and *timeout* combined, and once separated:

Algorithm Each algorithm is mined individually to identify patterns which may be exclusive to the algorithm.

Group All algorithms in their respective groups *math*, *sort* and *neural network*. The goal is to identify if patterns can be mined over different algorithms, and if this would yield more generalizable patterns.

Total All algorithms of all groups are mined together, to see if patterns can be identified independent of a specific domain.

Similar to the bug mining experiment, all minings were conducted twice, to analyze the effects of *embedded* vs. *induced* mining. Both minings use the entire identified population in all experiments.

The experiment settings were:

Pattern size is limited to 10 for *induced* and 6 for *embedded*. This had to be done since several groups over the experiment consist of only one AST, making all patterns of the AST significant.

Hierarchy is a data-type-independent MiniC hierarchy, only mining two layers. Data-type-dependent, and abstracted (e.g. DoubleLiteral or DTLiteral). To account for different loops, *For* and *While* loops were combined to a general *Loop*.

Redundancy the strategy for redundancy reduction is *closed*, meaning only the largest pattern containing sub-patterns survives.

Metric Two metrics were applied. *Maximal contrast* and *maximal support*. Both were set to 0.8, or reduced to (n - 1 / n) if less than 10 AST existed in a group, to ensure that at least one outlier was always allowed. For *group* and *total*, both metrics were set to 0.4.

Growth For the *induced* minings, the 9,000 most discriminative patterns were grown every time the pattern size was increased. Due to memory limitations, this had to be reduced to 3,000 for the *embedded* minings. The sizes are vastly larger than in the bug-mining experiment, since the applied metric prunes more of the search space.

Top N return Only the top 15 patterns according to the discriminability metric were returned per group. This means 45 patterns for all algorithms with a timeout and 30 patterns for all others.

Pattern Verification

After the pattern mining, the patterns were again manually analyzed and transformed into machine-readable anti-patterns or patterns. Some selected patterns were verified manually and dismissed if they did not have an impact on run-time performance. This manual verification was conducted via the verification mechanism outlined in Section 5.3. This verification was attempted in two different ways.

Patterns that purely deal with one type of node, observed as pattern or anti-pattern over different ASTs, were analyzed via a specialized mutator. This mutator identifies all nodes of the given type, and replaces them, leaving the rest of the AST intact. The goal of this type of verification is to modify as few nodes as possible to only measure the impact that one node type has on run-time performance. This direct replacement happens over any AST that has been identified as successful in the KGGI experiment suite, and has a run-time that did not lead to a *Timeout*.

In some cases a direct replacement is not possible. In this case the same approach as during the bug-pattern verification is applied. One mutator enforces a specific mutation point and injects the pattern at this point. All attempts with this type of mutation are conducted only on the original 25 manually written ASTs.

The verification is done by performance profiling 100 ASTs. Only such ASTs will be profiled that succeed at all test cases. The only exception is if a AST produces a *Timeout*, which will automatically be counted as verification for anti-patterns, or invalidation for patterns.

The confidence that an anti-pattern is responsible for the run-time performance is measured by the amount of all timeouts plus all ASTs that increased in run-time.

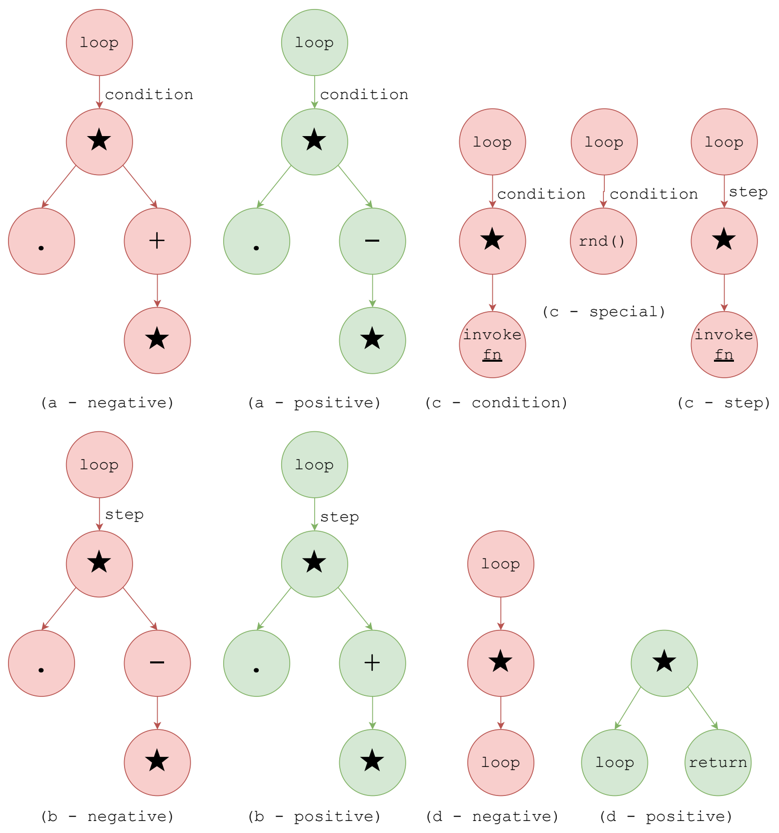


Figure 6.25: Performance patterns regularly occurring in loops. Performance is impacted negatively (a - negative) if the stop condition is set to a higher limit, or improved if it is reduced (a - positive). If the step size is reduced (b - negative) or increased (b -positive) this has the same effect. Often, invocations are used in the stop condition (c - condition) or step (c - step). All timeouts during benchmarking can be explained by a specialization of d, which is the call to *random* (c - special). In some cases, mostly due to the crossover operation in KGGI, loops are duplicated into each other (d - negative), sometimes this includes the a *return*, which has no negative impact on the run-time, but increases the amount of dead code (d - positive).

The confidence that a pattern is responsible for improved run-time performance is the inverse, e.g., the amount of all ASTs that had a decrease in run-time performance.

Identified Performance Patterns

Loop Related Patterns

Most of the performance patterns are tied to manipulation of the loop conditions or step variables. All identified patterns are shown in Figure 6.25. The shown patterns are the manual reduction of a multitude of ways these manipulations have been conducted. Different adaptions have been found in almost all algorithms and groups, in both embedded and induced mining. The loop patterns, working together with other patterns explained below, are the core reason for almost all performance improvements recorded.

First, the stop condition for a loop (a) is either raised, representing an anti-pattern, or reduced, representing a positive pattern. Especially in the case of *sort* algorithms, *raised* can also mean that a specific end condition has been increased, since they often count down.

A rather similar pattern (b) does the same for the step size. Often, the step size is dramatically reduced by conducting various arithmetic operations with the step variable or other variables manipulated in the loop body. The step size can also be increased, improving the run-time behavior. This is also the reason for the drastic performance improvement of *inverse square root*. Due to an oversight by the author, the inverse square root has

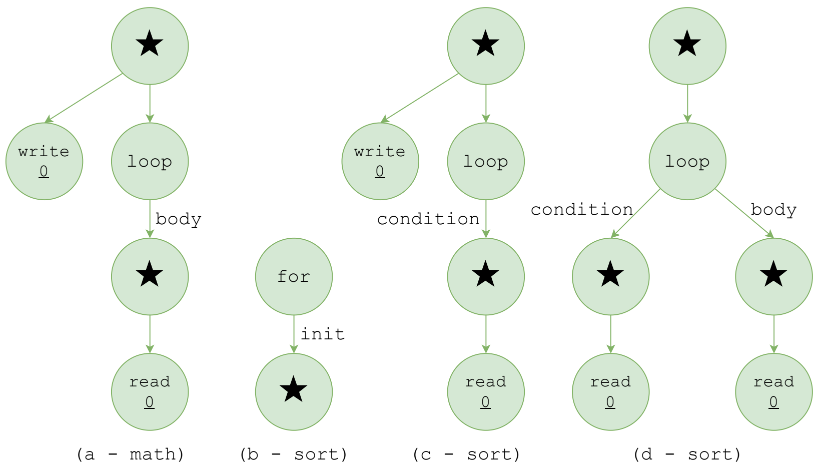


Figure 6.26: Performance patterns in *math* and *sort*. For *logarithm naturalis* the initialization condition was improved to a better guess in several ways (a - math). The sort functions had their initialization condition (b - sort), or a specialized variable for breaking (c, d - sort) modified.

a too high loop limit, since the final operation $return 1/x$ reduces the reachable accuracy. The best solution for *inverse square root* simply halves the amount of loop iterations (see Figure 6.22) as shown in Listing 6.3.

```c
float invSqrt(float x) {
    float result, h;
    int i, tablePosition;
    result = x;
    for (i =  1 ;  i + i + 1 < 50 ; i = i + 1) {
        h = (result * result - x) / (2 * result);
        result = result - h;
    }
    return 1 / result;
}
```

Listing 6.3: Inverse Square root optimized by reducing how often the loop is executed.

This is often done in rather nonsensical ways, which is usual for results from genetic algorithms [111]. These include unnecessary invocations (c) in both the condition or steps, which can be considered a specialization of the first two patterns (a,b). One further specialization is the call to *random()* (c - special) in the condition. This pattern is responsible for all timeout exceptions during benchmarking. The reason for this is that random in C returns a value between 0 and 32,767, which rarely hits the equality checks, or less than checks. This has a negative impact on run-time performance, but since the likelihood is high that the loop is conducted often enough to reach the desired outcome, most ASTs passed the semantic validity checks.

The final group (d) occurs, likely because of crossover operations. A loop is a child node of another loop, often conducting the same thing, such as the entire training phase of *neural networks* conducted as a loop inside the already existing training phase loop. This also sometimes occurs in *efficient* solutions, but in this case the entire block is moved over, including the return statement. This produces code bloat, with a large part of the AST never executing, but does not impact the run-time negatively.

Patterns Exclusive to Algorithms or Groups

Some patterns are exclusive to one group. In the *Logarithm Naturalis*, the performance improvement stems from an improved initial guess (Figure

6.26 - a). While it was not explicitly found in *Logarithm 10* it is assumed that the performance improvement there is for the same reason. This cannot be generalized into a pattern, but indicates good search positions for math functions in GI. Finding the improvement of the start condition as a pattern is interesting, since there is existing research in GI which deals with the generation of good values for lookup tables [103, 104].

The remainder of specialized patterns are exclusive to the *sort* tests. It always involves the manipulation of the core variable driving the sort implementation. This was found in *merge sort*s *lo* variable, which is also used in the variable initializer of the for loop (b). In *shell sort* the h variable is initialized before the main loop (c), an example of this is given in Listing 6.4. For *heap* sort, this concerns the *index* variable (d) which is manipulated during the loop body and accessed in the condition. In *quick sort* and *quick sort inlined* the *top* and *p* variables are often manipulated. This is also the reason for the large amount of timeouts in those two algorithms. The outcome for sort algorithms is the same. Due to smart manipulation, the run-time performance is improved significantly. However, if the sort was benchmarked with multiple inputs this might not always improve run-time performance.

Listing 6.4: Shell sort is improved by changing the initialization of h to 1.

```
array shellSort(int x[], int len) {
    int h, i, j, cont, tmp;
     h = 1;

    while (h >= 1) {
        // unchanged shell sort
        ...
    }
    return x;
}
```

General Cases

The following patterns, shown in Figure 6.27, mostly occurred in the mining attempts that consisted of all algorithms in a group. They also rarely occurred in specific algorithm settings.

Division (a) and **multiplication** (b) as operations occur in most ASTs. However they occur far more often in the *inefficient* groups, especially in *math* and *nn*. This is reversed for *sort* where both occur primarily in the *efficient* group. Similarly **invoke** (c) frequently occur in the negative groups, again the exception being *sort* where invocations regularly occur in the positive space as well. The exception is **chained invocations** (d), which are exclusive to the negative group. Most often two chained invocations are identified, in specific algorithms they can also occur as three chained invocations. The reason for all three is likely the same. They are most likely symptoms of the already discussed patterns concerning loops. All the patterns frequently occur as children of loop conditions or variable increments. The likely reason why *sort* shows positive behavior on the invocations is that some sort algorithms call helper functions that were not inlined (quick sort and merge sort).

The final pattern that occurs frequently over all groups is **double** (e). This is the only discriminative data type that was identified on a regular basis.

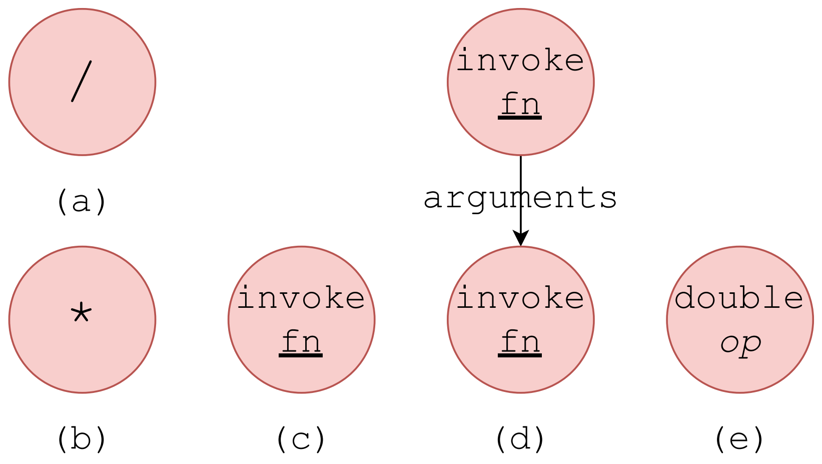


Figure 6.27: Patterns occurring regularly in most mining experiments as well as the combined groups. The division (a) operation and the math (b) operation are identified as negative in *math* and *neural network*. Invocations are similarly often discovered as negative (c) and in some cases chained invocations as well (d). The double data type (e) is discovered generally independent of the operation (addition, division, ...) as negative, except for the *sort* operations.

Aside from string and char, which were excluded after the bug mining experiment (and are still in rare cases identified as discriminative), double is the only data type not used by the original algorithms that was not excluded. Double is most likely another side effect, being discriminative only because of where it occurs. Similar to all other patterns discussed in this section, the double data type is overwhelmingly negative in *neural network* and *math*, but mostly positive in the *sort* algorithms.

The pattern is fairly easy to attempt to verify, by replacing all double nodes with their equivalent float version. Casts from other data types to double are similarly replaced with casts to float, and cast from double to float is simply removed. When attempting to prove that **double** was responsible for the run-time performance changes, the results are inconclusive, with a confidence of only 55% that the AST is more performant if using **float**. Attempting to identify how much the run-time performance changes is similarly inconclusive. The run-time performance increases on average by +1%, with most outliers in the *sort* algorithms (+2% increase overall). One outlier reduces its run-time by -37%, while others increase their runtime by about +20%. The *neural network* suite ranges between + 6% and -3% with a change of +0.14% overall. The math algorithms are the only ones slightly improved, with -0.39% overall. This indicates that **double** is in most cases irrelevant, as changes of only 1% or less is likely just a statistically insignificant measurement noise. The larger outliers may indicate that some double rounding errors are actually the cause for the behavior in the loop stop conditions or increment statements.

Outlier Cases

The following patterns mostly concern outliers or patterns that were only observed once over all experiments. These include some specific patterns that were analyzed, such as the neural network *neural network - swish* or *math - inverse square root*.

Variable Type Switch

One outlier in *math - square root Java* showed the pattern visible in Figure 6.28. This pattern was only discovered since this is the only outlier in the

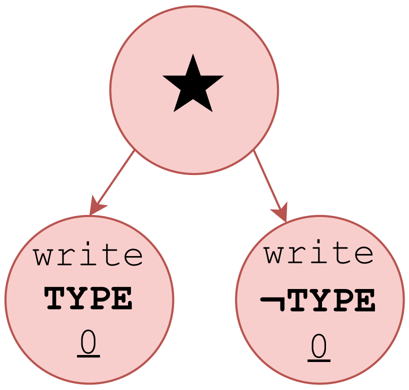


Figure 6.28: Multiple writes of different data types to the same variable have a negative impact on run-time performance.

algorithm group, meaning that the entire AST is discriminative. As the only difference between the original AST and the outlier is that first a char write happens to variable x, and then a float write happens also to variable x, the anti-pattern was discovered. This is shown in Listing 6.5. Due to the change of the primitive type that the frame slot *x1* has, it is likely that Graal cannot optimize the frame slot and thus has a much worse run-time performance than the original solution.

Listing 6.5: The only outlier in square root Java changes the type of the variable x.

```
float sqrt_java(float x) {
    char x1 = (char) 0.2;
    x1 = x;
    return sqrt( x1 );
}
```

This might be the only generally applicable pattern identified which is directly applicable to compilers. The pattern is irrelevant for strongly typed languages such as MiniC. The issue only occurred since it can be represented in the AST and was created via KGGI, but the parser would not allow a manual creation of such code. In dynamically typed "programming languages" such as Python or JavaScript this may be a relevant pattern to look out for though.

Neural Network - Swish Outlier

The outlier in swish essentially uses an exploit to optimize 99.83% of the run-time away. The simple outcome, as shown in Figure 6.29, is that one single outlier AST returns the output. The actual AST is more

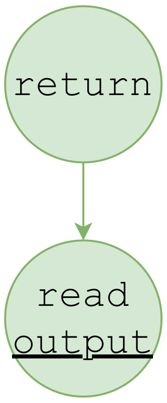


Figure 6.29: The *swish* outlier skips the training phase and simply returns the training output.

complex, and the discriminative pattern that was found actually shows an assignment of the training-output argument of the function call to the output variable. The implementation simply replaced this assignment with the training phase, meaning it actually conducts this assignment 1,000 times, before then applying the neural network's uninitialized neurons to the output data. As they are untrained, no weights were assigned and no changes to the output are made, returning the unchanged output data, as shown in Listing 6.6.

```
array nn_swish(int numTrainingSets, float output[],
float training_inputs[][], float training_outputs[][])
{
    // prepare variables
    const int numInputs = 2;
    ...

    // train
    for (n = 0; n < 1000; n = n + 1) {
        // output assigned from training output
        // training phase is removed from loop
         output = training_output;
    }

    // validate results
    for (x = 0; x < numTrainingSets; x = x + 1) {
        ...
    }

    return output;
}
```

Listing 6.6: Swish returns the training output instead of training a neural network.

This essentially has the same outcome as the pattern shown in Figure 6.29. No matter how many test cases would have been used, the neural network would always return the expected output. This shows a flaw in the design of the experiment, and the test should have actually called the already trained network with different tests than were used for training. Another lesson that might be taken from the outlier, is that there are likely easier ways to implement an XOR gate than training a neural network, as the output has exactly that gate encoded.

Summary

The results of the experiment for mining performance-patterns has similar indications for this research as the bug-mining experiments did. Most patterns in the experiment were identified with *induced* and *embedded* mining. Both minings have their merit. *Embedded* patterns are usually more compact, and can identify locations that are far apart from each other in the AST. *Induced* patterns typically show not only the pattern itself, but also the location where the pattern most often occurs, but have more nodes than *embedded* patterns.

All identified patterns can be applied to KGGI either with specific goals of where to optimize, such as the loop condition, or initial guesses for math functions. The generally occurring patterns, such as division, multiplication and invoke do not produce verifiable results. This either means that they are simply co-located patterns with the real reason the AST was *efficient* or *inefficient*, or that there are larger patterns in relation that need to be applied for a general optimization pattern.

The only pattern that may be generalizable to a compiler is the check if a variable's type changes on the stack or heap (see Figure 6.28). This only concerns languages with dynamic typing, which MiniC is not. It also stands to reason that languages with dynamic typing will already have such a performance improvement in place. Nonetheless, attempting a reproduction of the pattern in a dynamically typed language may be interesting future work.

6.6 Threats to Validitiy

There are several threats to the validity of this work, concerning the identified patterns, performance measurements and verification of bug- and performance patterns.

The first threat to validity is the approach of testing. It is the selected form of semantic verification of all ASTs as well as measuring their run-time performance. This work uses the same approach as many other works in the domain of GI and GP. For the *math* and *sort* benchmark suites, a sufficient amount of tests has been selected. From the outlier discovered in *neural network - swish*, it becomes clear that the neural network suites may be insufficiently tested with just one test case. In this instance, multiple test cases would not have changed anything, but rather a split between training and test data should have been applied as is usual when training a neural network. This had to be done as a tradeoff between the available hardware and finishing the experiments in a sensible timeframe, but puts in question the results of the third experiment suite. The same can be said for the performance analysis. All performances were benchmarked once, instead of with multiple different inputs. This is again not unusual for performance evaluations [70, 107–109, 112]. Work in compilers compare themselves on benchmark suites instead of single cases [58, 59, 62]. This work shows a significant performance improvement over most attempted algorithms, similar to how compiler related work shows their comparisons over benchmarks. However, for each individual algorithm, there is no guarantee that it will not perform worse than the original AST on a different input.

With the performance measurements, there is a threat to validity on the hardware and the compiler. As all tests were conducted on the same hardware; the performance distributions and measurements may look differently on other hardware. The following settings applied:

Compiler Graal version 21.1.0 with Java 11 (11.0.11 64bit) were used.

Operating System Ubuntu 20.04.3

CPU A x64 12 core, 24 thread 4,600 MHz CPU was used. AMD Ryzen 9 3900X

RAM 128 GB DDR4 3,600 MHz. During pattern mining, at most 120 GB were used. For KGGI each single AST evaluator was limited to 128 MB.

The final threat to validity stems from the verification of the identified patterns. In this verification process, side effects are often unavoidable. If, for example, it has been identified that the *division* operation has a negative impact on an AST, ASTs must be modified to either remove such an anti-pattern (expect better run-time performance) or inject the anti-pattern (expect worse run-time performance). To maintain semantic validity, this may require modifications of other nodes as well. This makes it hard to determine if the *division* was the true reason for the run-time performance decrease, or if the nodes often occurring together with this operation are really at fault. In this work, this threat was attempted to be mitigated by proving patterns only via mutation instead of addition wherever this was possible. This was done to ensure that negative impact is not only measured by new code that has to be executed.

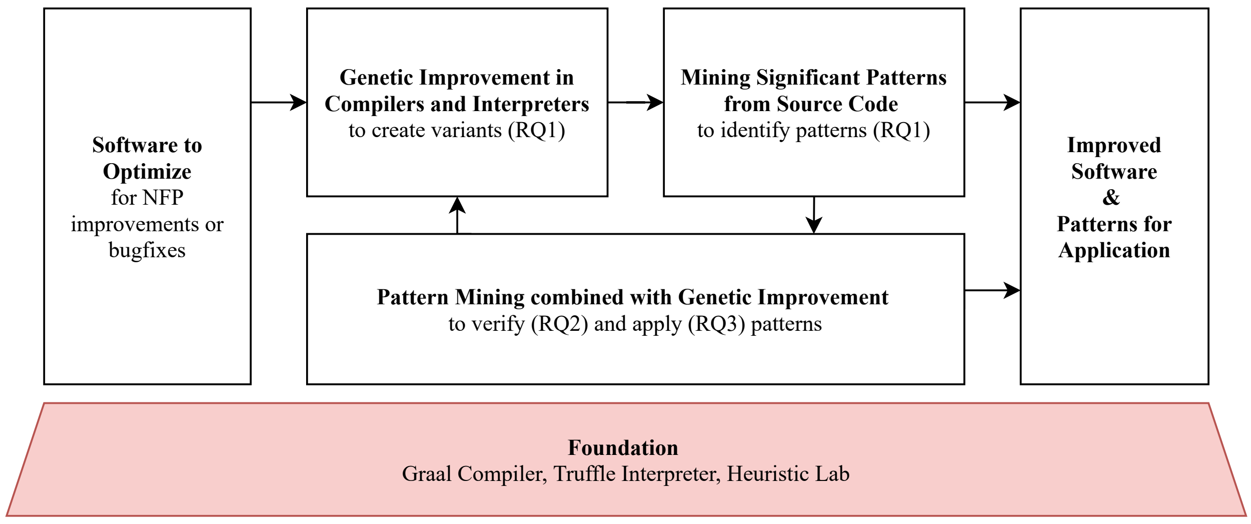
Related Work 7

This work combines and expands upon algorithms, techniques and approaches from several distinct research fields. The following sections summarize the state-of-the-art in these fields, and show how this thesis distinguishes itself from existing work. To achieve this, related work is analyzed in the following contexts:

Pattern mining in source code summarizes work that attempts to find recurring patterns in source code. The work also categorizes recurring themes, such as the approach taken (dynamic, static, hybrid), representation mined (graph, tree, sequence), and domain being mined (bugs, patterns, clones, ...). As a comparison to Independent Growth of Ordered Relationships (IGOR) relevant algorithms in this area are also discussed.

Genetic Improvement and Genetic Programming discusses work that also attempted to utilize or identify patterns for or via Genetic Improvement (GI) and Genetic Programming (GP). As a comprehensive survey on GI already exists, this analysis does not attempt to categorize the related work as done in pattern mining, but summarizes the survey by Petke et al. [29]. Some more recent work as well as work that was not in the scope of the study is discussed in addition.

Code Optimization for Compilers and Interpreters takes a look at comparable techniques to GI that attempt to modify source code to provide additional benefit to existing compiler optimizations. Pattern-related approaches for compiler optimizations are discussed as well.


Software to Optimize
for NFP improvements or bugfixes
Genetic Improvement in Compilers and Interpreters
to create variants (RQ1)
Mining Significant Patterns from Source Code
to identify patterns (RQ1)
Pattern Mining combined with Genetic Improvement
to verify (RQ2) and apply (RQ3) patterns
Improved Software & Patterns for Application
Foundation
Graal Compiler, Truffle Interpreter, Heuristic Lab

7.1 Pattern Mining in Source Code

There is a wide variety of work on mining software on different topics, such as mining the process of developing software [113], coding conventions by developers [114, 115] or class relationships [116]. The following, summarizes only such work that attempts to detect recurring patterns in software, either with the target to localize defects, or to localize recurring changes.

Work attempting to find known patterns in existing programs is also not discussed, as it is not the focus of this work. Deniz and Sen [117] provide an overview of the application of machine learning techniques to identify patterns in the domain of multithreaded applications.

Defect prediction is a similar field, primarily utilizing neural networks, semantic features and representations in the form of Abstract Syntax Trees [118]. The field deals with answering if code is buggy, not where the bug may be located. Patterns are also not identified. Thus, this work is of scope for the work presented here.

Figure 7.1 shows the general categories the related work is analyzed in. This encompasses the representation of source code used in the mining process, the domain the mining is done in, and the approach to mining being applied. The approach is split between dynamic approaches that use information gathered at run time of the program, and static approaches which only consider the source code itself. Hybrid approaches use a mix of both.

Nguyen et al. [44] present a graph mining approach to detect semantic code change patterns. They mine 5832 GitHub repositories and conduct an in-depth analysis of 88 of these repositories, with several million lines of code. The mining utilizes a Program-Dependence Graph, e.g. a graph that contains both the control flow and the data flow at the same time, covering the semantics of the code in addition to the syntax. From mined code changes, they create a pattern consisting of a before-change graph mapping to an after-change graph. These patterns are then mined iteratively by growing the patterns and matching them to each other via isomorphism. They achieve a high performance by utilizing a greedy function, where only the most frequent extension of a pattern is considered during the growth phase. Previously, the same authors have mined patterns only via the control flow [47]. This work has later been extended into a statistical language that enables suggestions for code completion with 75% accuracy. In their work they also show the importance of normalization of ASTs, i.e. labeling variables and literals as well as special values such as null and empty strings or the number zero [48].

Balanyi and Ferenc [50] start off a research series where they define the Design Pattern Markup Language (DPML) which is an extensible markup language (XML) based description for design patterns. The core focus of their work is mining patterns that overarch multiple classes. The patterns are grouped into creational patterns (i.e. patterns creating objects), structural patterns (i.e. patterns in the composition of a class or object), and behavioral patterns (i.e. covering the interaction between classes). In later work [119] they introduce a graph-based refactoring for C++ source code. They also show, via the mining of anti-patterns, that there is a strong correlation between anti-patterns, bugs and the maintainability of software [51, 120, 121]. This is done via the Columbus pattern mining tool which utilizes a Language Independent Model (LIM), a graph format representing classes, methods, attributes and other source code representations. This model is enhanced with an additional graph containing various source code metrics correlated via decision trees with anti-patterns.

Oßner and Böhm [38] discuss mining defects in multithreaded programs. They introduce concepts to extend Dynamic Call Graphs with temporal edges that allow the representation of calls over multiple threads. They

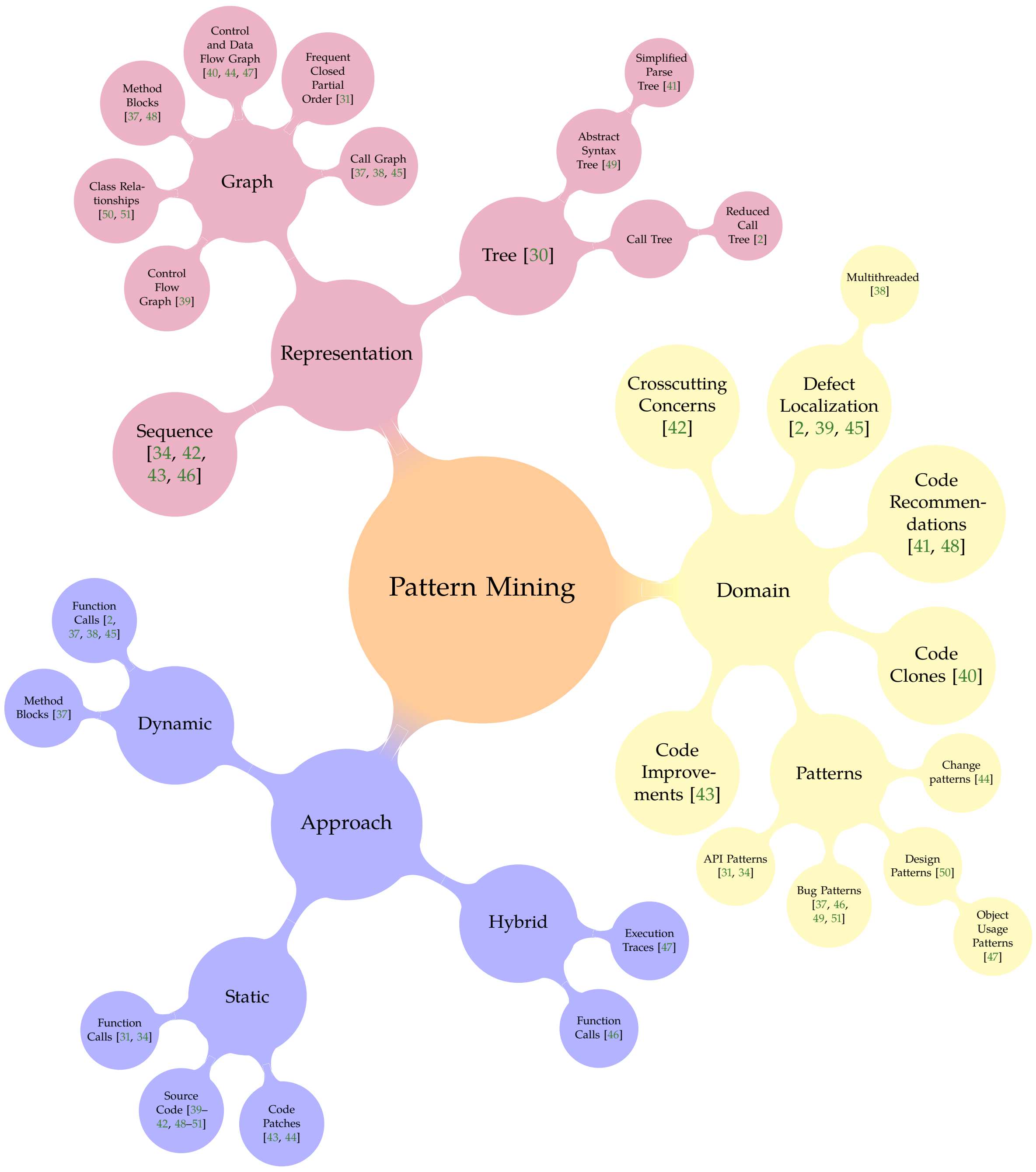


Figure 7.1: Related work in pattern mining. All related publications are categorized in the used source code *representation*, *domain* the approach is applied in, and *approach* used to gather the information being mined.

categorize call graphs in two separate sets, failing and succeeding. Via distinct pattern mining, they use the Information Gain metric with closed frequent subgraph mining to identify suspicious subgraphs and manage to identify multiple concurrency bugs. Like other dynamic approaches, their work has to deal with so called Heisenbugs [122], bugs that disappear due to their approach of instrumenting.

Acharya et al. [31] mine API pattern usages from source code. This is a widely researched domain that is only slightly related to the mining of source code presented here. [31] is interesting as related work as the authors conduct a static analysis of function calls, i.e. they analyze the source code and create call sequences from it. These sequential traces are then analyzed with frequent subsequence mining and are combined into a Frequent Closed Partial Order Graph, expressing frequent call orders of functions in a program. The work shows the issues of static analysis of code which produces many infeasible traces that won't occur during execution, but also advantages of scalability and removal of set-up costs as no code needs to be executed or made executable.

Qu, Jia, and Jiang [40] introduce a staged mining approach to find cloned code in software systems. They apply a spatial pattern search on a program dependence graph (PDG) via rearranging and encoding. Only on matched patterns that have been detected via the spatial search, a second graph-based approach is used. This second step uses isomorph graph mining, and thus greatly reduces the amount of false positives that would occur from only using spatial search. [40] compare themselves to five other approaches, showing that they need less manual preprocessing in the source code to attempt mining, and also find significantly less false positives (1 vs. 7 of the next best) and false negatives (2 vs. 10 of the next best), while also having a lower average run time. Their combined approach reduces the inherent issues of the NP-complete subgraph isomorphism, which requires exponential runtime to solve, while also reducing the large amount of false positives that spatial search produces.

Luan et al. [41] also follow a staged mining approach. The authors implement code recommendation using structural code search in four different programming languages. They use a simplified parse tree, similar to an AST, that is based on keyword tokens, and simplification of variables and interactions between them. From this they conduct a featurization on which an initial highly performant search for structurally similar candidates is conducted. In a second step, candidates are pruned via a greedy algorithm and are re-ranked for similarity with the original code. The ranked candidates are then clustered into groups and intersected with each other to provide meaningful snippets around the originally searched code. Even though [41] utilizes a tree representation, the algorithm does not use pattern mining. For both the pruning and the intersection of snippets, a greedy algorithm is utilized that can lead to a loss of candidates but is more efficient than frequent subgraph mining. [41] also determined a good similarity score threshold for meaningful patterns. This was done by manually labelling code snippet tuples to be similar or non-similar. From this, they applied their scoring algorithm to the manually labelled items, showing that there is a significant statistical difference in both groups.

Ishio et al. [42] apply mining of sequential patterns to identify cross-cutting concerns in software. For this, they modify source code into a sequence of tokens. For example, an if-then-else block with opening and closing brackets is resolved into IF, ELSE, END IF. This sequence is then mined for patterns occurring in sequence, focusing on method calls that occur together. The limitations of a sequential approach are the size of the found patterns, and, due to the mining approach, that patterns occurring often in fewer methods may get filtered out. In later work [43] the sequential mining is expanded by more normalization, enabling the mining of code improvement patterns and their trends over time by comparing code patches.

Wang et al. [34] introduce two quality metrics for mining API usage patterns. These are succinctness, i.e., explaining usage with as few patterns as possible, and coverage, i.e., explaining as much of the API as possible. They improve both of these metrics by a staged mining approach similar to [41]. In a first Mining step, the BIDE algorithm is used to mine frequent closed sequences. These are then clustered with n-grams to measure similarity of sequence tuples and to cluster them around centroids with a minimum similarity threshold. These clusters are then consolidated into patterns. While the mining approach appears on sequences, the resulting patterns are shown to developers as probabilistic graphs of which path in a source file is likely to be taken.

Hanam, Brito, and Mesbah [49] utilize an unsupervised machine learning technique to discover bug patterns in JavaScript. This is done by mining change commits from GitHub and comparing the original and the changed ASTs of the source. These are then converted into a feature vector based on the code size (lines of code) and basic change types defining changes and context. Based on these features, a clustering and ranking is conducted. The patterns are not grouped into an AST etc. but rather analyzed manually.

Livshits and Zimmermann [46] introduce DynaMine, a tool to detect patterns and pattern violations in source code. This is done via a combination of static and dynamic code analysis. Source code revisions are mined into sequences of function calls, of which patterns are mined. Interesting patterns selected by a user are then instrumented and executed, recording violations in the process.

Di Fatta, Leue, and Stegantova [2] introduce discriminative pattern mining, i.e., the concept of splitting the data set being mined into multiple sets of failing executions, passing executions and specific neighborhoods. Their approach utilizes Reduced Function Call Trees, which contain the function call traces, and reduce multiple calls of the same function to 0, 1 or 2, whereas 2 stands for any amount of calls. Based on this approach, they identify suspicious function call patterns and apply a ranking on each function implying the likelihood that the function is containing a bug.

Cheng et al. [37] also utilize discriminative pattern mining. They do so on two different representation levels, the call graph between functions, and the basic blocks of a method. Even though their approach does not mix these representations but rather mines each separately, they call this representation a Software Behavior Graph. In their work, they show the high performance of Discriminative Graph Mining via LEAP search [36] as the approach ranks patterns via the Information Gain metric and only continues growing the search space for the top n-ranked patterns. Their work shows that the representation granularity is important as they find different bug patterns via call graph or basic block.

Liu et al. [45] present a classifier to detect non-crashing bugs, i.e., bugs that are identified through a failed test instead of a run-time exception. This is achieved by conducting discriminative subgraph mining on a given set of correct and incorrect runs, represented as a graph of function calls. They utilize mined frequent subgraphs in a Support Vector Machine (SVM) to classify subgraphs that are likely to be the location of a non-crashing bug. The subgraph mining itself is done via the CloseMine algorithm, an extension of the CloseGraph algorithm, by adopting a naive search order that enables skipping parts of the search space to speed up the mining process.

The presented related works show that software has been mined for patterns in most conceivable representation forms with different approaches. The discriminative pattern mining approaches are the closest to our work, which is also based on discriminative pattern mining, extending it to multiple categories. The primary advantage of our approach over the related work is the application of a mining with a representation that is native to compilers and interpreters, namely Abstract Syntax Trees (ASTs), in addition to using information directly available from the compiler and interpreter. This work also shows how these patterns can be verified with a confidence metric, in both the functional and Non-Functional Property (NFP) domain of run-time performance, which is the first of its kind.

Algorithms

Frequent Subgraph Mining Algorithms are generally categorized into two core concepts. The Apriori algorithms generate all subgraphs of size n, select relevant subgraphs according to a minimum support threshold, and then generate all subgraphs of size n+1 containing these selected subgraphs. Pattern Growth algorithms work on a similar basis, but instead extend only frequent subgraphs until the subgraph becomes infrequent. In general, pattern growth algorithms perform better than apriori algorithms, as they only extend the search space in relevant areas [87]. The following shortly summarizes the algorithms shown in Figure 7.2 used in the studies above. Such a summary is also available by Jiang, Coenen, and Zito [123], albeit with a broader view on the topic. Our summary has a focus on algorithms used in the domain of source code mining.

Agrawal and Srikant [32] introduce the original Apriori algorithm in the context of mining association rules in large databases. Many adaptions of this algorithm exist, such as the Apriori-based Graph Mining (AGM) algorithm [129].

Ester et al. [53] define a clustering-based approach to mining recurring samples in a database. This is done by calculating a density according to which graph data is clustered. This does not discover frequent subgraphs, but is an alternative way to discover patterns, as was done by Hanam, Brito, and Mesbah [49].

Asai et al. [33] introduce the FREQT algorithm, an extension to the Apriori algorithm to mine frequent-order tree patterns. The algorithm improves its efficiency by always conducting a rightmost expansion. Pham et al. [130] expands this work with FREQTALS introducing additional constraints. They require a minimal pattern size to ensure that a pattern is sufficiently large to be usable and to provide it in a context the pattern can be applied in. Additionally, they consider the type of nodes, i.e., the labels such as TypeDeclaration or Block, to exclude patterns that naturally occur due to a languages' grammar but do not provide developers with new information. These labels are also used to enforce valid root nodes for a pattern, or for pruning irrelevant paths.

Yan et al. [36] define the LEAP algorithm, which works by taking advantage of the fact that significance metrics such as information gain

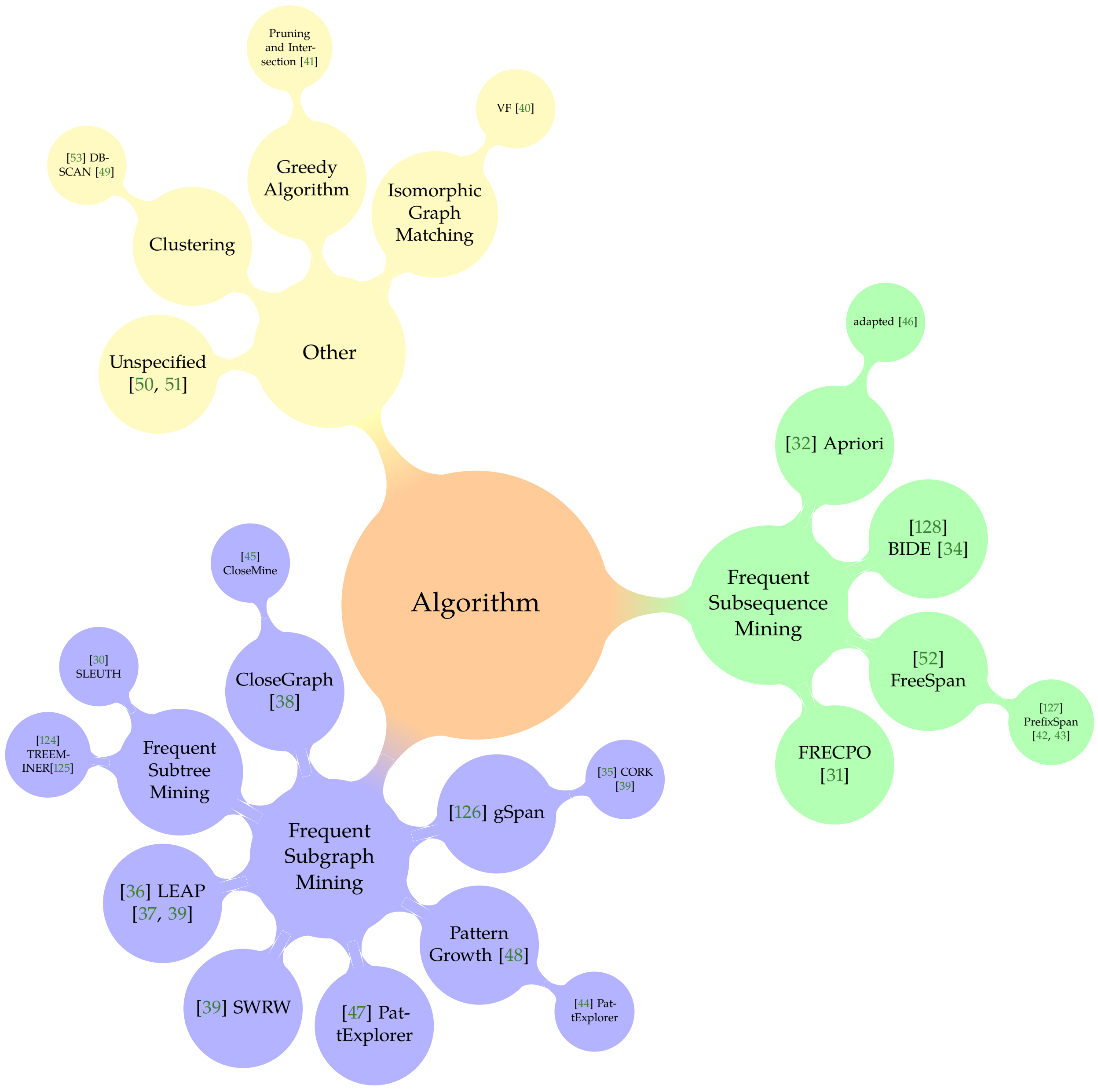


Figure 7.2: Algorithms discussed in pattern mining. The citation in front of an algorithm name defines the algorithm, while citations after the name identify usages of the algorithm.

correlate with the structural similarity of patterns. They provide two concepts to use LEAP. sLEAP (structural leap search), and fLEAP (frequency descending mining). In their work, the authors show that LEAP outperforms branch and bound algorithms.

Henderson and Podgurski [39] introduce, Score Weighted Random Walks, for localizing faults in code. This is done by providing two groups, passing and failing tests, and constructing control flow graphs in an algorithm for the test sets. Both of them are mined by random walk with several metrics enabling the guidance of the algorithm to find sub-graphs that show a difference between the two test groups. The algorithm outperforms other subgraph-mining algorithms such as CORK [35] and LEAP [36] and find more relevant patterns for behavioral fault localization.

Nguyen et al. [47] introduce PattExplorer, which follows a simple pattern growth process by mining patterns of size 1 and then iteratively increasing the size by 1 only in patterns that occur frequently. They tackle graph isomorphism via vectorizing the nodes and edges in a graph.

Yan and Han [126] define the graph-based Substructure pattern mining (gSpan) algorithm, which is neither an apriori nor a pattern growth algorithm. gSpan instead works efficiently by mapping all graphs into an encoding built up by depth first search. It then uses these encodings to identify frequent subgraphs via a depth-first search, instead of iteratively growing the entire search space.

Thoma et al. [35] define a correspondence-based quality criterion (CORK) that integrates with gSpan, presenting a speed-up of gSpan and improving the accuracy measure of presented discriminative subgraphs this is done by selecting and utilizing features to be used for expanding the patterns. It has some disadvantages as well, such as requiring smaller pattern sizes to be pre-mined to produce the speedup and requiring minSupport to be sufficiently high to provide a payoff between the feature calculation and the reduction produced by those features.

Wang and Han [128] introduce the BI-Directional Extension (BIDE) algorithm. It mines frequent closed sequences as opposed to mining frequent patterns. A closed sequence is a sequence that is maximal in the sense that no larger enclosing sequence exists with the same support. The advantage of closed sequence mining is that the redundancy of patterns is deceased compared to mining all frequent patterns. BIDE achieves this efficiently by pruning the search space via backwards extension after the growth phase.

Pei et al. [127] define the PrefixSpan algorithm, which is an extension of FreeSpan [52], an algorithm devised by the same authors. FreeSpan mines patterns in sequences by using a projected sequence database that guides the search and reduces effort during sequence generation. PrefixSpan expands upon this idea by using a prefix list, where each prefix in the list has its own projected postfix database.

Zaki [30] defines the SLEUTH algorithm, an improvement of TREEMINER by the same author [124, 125]. Among all discussed algorithms, SLEUTH is most comparable to IGOR. SLEUTH is a pattern growth algorithm specialized for the mining of trees in an embedded unordered way. While it conducts mining in an unordered way, which in general is more run-time intensive than ordered mining, SLEUTH is still highly performant even though it generates all possible subtree mutations. This is achieved by an efficient string encoding of the patterns, which allows for fast injection during the growth phase, combined with a scope list of possible extensions stored in a database.

The primary differences between SLEUTH and IGOR are that IGOR supports both induced and embedded mining. SLEUTH on the other hand supports unordered mining, while IGOR only supports ordered mining. As shown in Chapter 6 ordered mining is important for identifying patterns on the granularity of AST representations, since the relationship types (e.g. loop condition vs. body) are relevant.

7.2 Genetic Improvement and Genetic Programming

GI in the context of this work was selected as a basis to identify patterns in source code ^{RQ1}. This was done explicitly, as GI often attempts to improve the NFPs of software. As these NFP are typically part of the fitness function, this also allows our work to be based on accurate measurements, working on executable code, instead of inferring these values second hand, as would be possible by mining software repositories and attempting to find patterns in this area. Alternatives in this goal would be synthesis approaches [131, 132] or program translation approaches [133]. Synthesis approaches usually attempt to generate new code instead of improving upon existing one. Program translation attempts to map a program from an input to a similar output. These methods do not provide the desired results, which are multiple versions of the same program with different ASTs that allow the mining of recurring patterns influencing the NFP.

RQ1: How can recurring patterns be identified that impact or improve a functional or Non-Functional Property?

Related work in GI is summarized by Petke et al. [29] in "Genetic Improvement of Software: a Comprehensive Survey". The survey identifies 3132 publications between 1995 and 2015, analyzing 66 core GI publications in detail. Before discussing some additional publications, a summary of this survey is provided. The survey outlines the origins of Genetic Improvement, beginning with Program Synthesis and GP moving via Search-Based Software Engineering (SBSE) towards GI. This shift primarily relates to not generating new functionality (GP), but instead improving or repairing already existing functionality (GI). What remains similar over the domains, and the 20 years analyzed in the work, is that testing and validation remains the primary method of validating semantic correctness. An interesting find of the work is that modifications by GI consist of fewer than six lines of code, often being minor modifications. This also resembles the results of our work, with many anti-patterns preventing mutational-bugs in the compiler consisting of just a few nodes.

Petke et al. [29] in their survey defined their search by four primary requirements. 1) that metaheuristic search is used (not necessarily a genetic algorithm), 2) that variants that do not preserve semantics can be produced during the search, 3) that existing software is used as input and 4) that modified software seeks an improvement over existing software. This also summarizes the goals of GI quite well. The trend of GI not necessarily utilizing genetic or evolutionary algorithms continues past

GIN: GI in No Time https://github.com/gintool/gin

1: Python General Framework for Genetic Improvement https://github.com/coinse/pyggi

the survey's scope. For example, the GinTool GIN utilizes a local search algorithm [70, 109], and so does the PyGGI[1] framework [107, 108].

The fitness function utilized in GI is in all cases of the 66 core papers utilizing test cases, though the granularity is utilized differently. In the simplest case it is just counting the passing or failing test cases, while more fine-granular functions instead use a distance metric from the desired results [29]. This is also the approach taken in our work. Petke et al. [29] identify an open issue in the characterization of test suites, in how to best utilize them to guide towards improvement of NFP. However, a correlation exists between coverage and the successful identification of software repairs. In our work, utilizing the fitness functions was attempted to continuously increase pressure on the fitness function, and thus on the search space, though the results were not promising (see Section 3.3).

The primary types of applications identified by [29] focus on software repair, e.g., fixing bugs. Program correctness is in this work often considered a NFP. Run-time performance is the second largest area, together with software repair, making up almost two thirds of all papers in the study. The remainder of work deals with parallelization, energy consumption, generating new functionality from existing ones, such as evolving inverse square root out of square root [102], slimming down code size, reducing memory consumption and specializing code for specific hardware. Research in all areas rarely uses multi-objective optimization, which is an open issue in the domain, instead combining all fitness values that are attempted to be optimized into one fitness function. Harder to measure NFPs have an issue in the fitness function, where research typically attempts approximation instead of actual measuring. Our work also utilizes a fitness function that does not combine multiple objectives. However, approximating run-time performance based on the given AST is problematic as the AST is modified by the compiler optimizations in a derived representation, and running every individual in the GI population for 200,000 repetitions, without influence from other threads, to achieve accurate measurements is too costly during the GI run. Thus, we chose to only measure the accuracy instead of measuring the run-time performance during the GI experiment. The run-time performance of all successful variants is measured after the execution of the entire experiment.

Only three different representation forms utilized in GI are identified by Petke et al. [29]. ASTs, or similar variants, are the most common form of parsed variants in GI. The other two variants are byte code and source code utilized as text. Considering the previous section on pattern mining, both of these representations would be treated as sequences. Graphs are not considered in any of the publications analyzed, also identifying possible future work.

Only one related work is known to the author, that deals with working on genetic improvement at the compiler level, which was published the same year as the first publication of this thesis. Yoo [134] published a position paper suggesting to apply GI directly at the level of the programming language, citing advantages such as ability to measure objectives, such as memory consumption, more easily and allowing more pervasive control of the program state. Further publications by Yoo introduce the

PyGGI framework [107, 108, 135], which allows line-level or AST-level mutations independent of a programming language. However, their work concentrates more on the language and not the compiler or the run-time environment. In comparison, our work uses access to the run-time-representations such as the stack and the heap, or to lower-level representations directly used by the compiler, such as the Truffle AST.

Mutation and Crossover Operations

There is a difference in how current research in GI applies its metaheuristics, compared to our work with Knowledge-guided Genetic Improvement (KGGI) (see Section 3.4). Our syntax graph that enables creation of an AST more closely resembles the traditional approach of creation using the mutation operation as seen in genetic programming [14, 68, 136], and earlier GI work [18, 137]. The *plastic surgery hypothesis* by Barr et al. [24] instead suggests *grafting donor code* from other parts of the software into the currently optimized piece of software. In a similar manner Schulte et al. [12] identified just three mutation operations applied to an AST, *copy*, *delete* and *swap*. They noted that using this mutation operation *37%* of mutants had no effect on the functionality of the mutated AST given a test suite that verifies the functionality, thus asserting the *mutational robustness of software*. These findings still seem to be the state-of-the-art, as more recent research still utilizes similar operator sets. GIN for example uses three edit types on lines, statements and constrained statements, all edits being of the groups *delete, replace, copy, swap* or *move* [70, 109]. These operators find their success because they produce fewer exceptions compared to previous research using GP methods. The GP methods found up to 80% of their individuals failing during compilation or at runtime [8, 9, 26], because mutations introducing new code often created exceptions such as attempting to read variables not yet initialized.

This poses the question of why we did *not* utilize the more recent mutation operations suggested by literature. The older type of mutations, replacing statements with newly generated sub-ASTs is a valid option. Schulte et al. [12] and Kinnear [106] both showcase their work on sorting algorithms ([12] expands with other algorithms). However, [106] manages this with mutating between 6 and 8 operators and operands in 4 different experiment setups. Likewise, Orlov and Sipper [26] successfully show their work on 6 experiments with 6-20 operators and operands available during the executions. This goes on to show that our form of mutation, while not being the state-of-the-art, is still a valid one. The underlying issue is the high failure rate of ASTs, and likely, why new types of mutation with a lower failure rate became popular. This is solved in our work via KGGI. As shown in Chapter 6, the introduction of anti-patterns and patterns can successfully lower the number of individuals failing, increasing the diversity in the population and reducing error rates. Modern GI relies less on the crossover operation, which naturally conducts the replace, swap and copy operations. In some cases, the crossover can delete as well, albeit only by replacement with a (smaller) AST. The advantages of these options are still existing in our research, as a crossover is still utilized. The major reason mutation and crossover is solved as described in the thesis, is that mutation purely attempting to move, duplicate or delete existing source code would be less useful to identify patterns. As most of the source code will relocate statements, fewer sub-ASTs will become discriminative, not allowing to identify novel patterns. In addition, the new type of GI closely resembles code motion, an already existing technique in compiler optimization, discussed in Section 7.3.

Previous Applications of Pattern Mining in GI and GP

Le Goues et al. [138][139] describe GenProg, a tool for automated software repair. GenProg itself works similar to other GP and GI frameworks such as GIN and PyGGI. It was designed to automatically fix bugs in software using genetic programming, utilizing ASTs and a fitness function based on passed and failed test cases. GenProg utilizes a genetic algorithm as search, with the mutator working on the statement level, either inserting, deleting or swapping statements from other parts of the source code. As crossover, a one-point crossover is used. The crossover happens at a weighted path, where the weight is defined as how often the statements are reached during execution. GenProg itself does not work with patterns. Kim et al. [111] compare their approach to GenProg. The authors note that GenProg can produce rather nonsensical patches that would not be accepted by developers. The authors instead manually analyze 60,000 human-written patches, identifying six core patterns that are responsible for 30% of bugs. They use these patterns in a Pattern based Automatic program Repair (PAR) approach, and go on to evaluate it on 119 bugs from open source projects. PAR successfully solved 27 of these, while GenProg only solved 16. Their work shows that pattern-based approaches can be more successful than GI itself, and also that their identified patterns are more accepted as they are hand-curated and thus understandable for developers. In our thesis, we hope to bridge this gap by identifying recurring patterns automatically from GI, filtering out nonsensical fixes in that way.

Jonyer and Himes [140] apply pattern mining during GP runs to find commonly recurring substructures in the individuals of the population that have a high fitness in regular intervals. The substructures identified as common are compressed into one node in the subgraph, to ensure easier re-use during subsequent crossover or reproduction. They do not apply mutation in their experiments. Patterns are used in their work to reduce the AST size to manageable levels, while also encouraging high performing substructures in the population. Their results show that this method reduces the needed amount of generations until a valid solution is found significantly.

This type of mining patterns to identify recurring substructures was also conducted by other authors. Langdon and Banzhaf [141][142] identify recurring patterns in an individual AST, instead of mining patterns from the entire population. They find that often the same sub-AST occurs multiple times in the same tree, and suggest that GP is successful in creating ASTs that are quite large but are made up of just a few repeated genes responsible for the fitness in the way they are put together.

Recurring substructures were also used as semantic building blocks by McPhee, Ohs, and Hutchison [143]. The authors do not trace the syntax of the given AST, but instead the semantics of what the AST does, in a boolean operator search space. With this they manage to find out that crossover operations have a high chance (approx. 75%) of not changing the semantics of the code at all, and thus not contributing to the search in the semantic space. This concept of building blocks being tracked by their semantics was also utilized to trace the genetic diversity and genealogy throughout entire GP runs [144–146]. The purpose of this ancestry graph was to analyze the impact genetic operators have on the search space and thus on the quality of the resulting AST. The work was also extended to trace and visualize the genealogy of building blocks through GP runs via a graph database [147–149], similar to how our work uses a graph database to store GI runs to later mine recurring patterns.

Tracing the genetic genealogy was also conducted by Burlacu et al. [150], to help get an understanding of how genetic fragments, e.g. sub-ASTs, are introduced into a search space and then propagated throughout the generations of evolution. The goal of their work is to use this genealogy information to guide the search of which building blocks should be preserved during the run. In this area of research, Burlacu et al. [151] also introduce a hashing algorithm that allows identifying ASTs or parts thereof. They currently drive the diversity of the population via these hashes by introducing a distance metric on the hash representations of the trees. In a similar vein, our work applies a bit representation of ASTs to identify a generalization or specialization between patterns (see Section 4.5).

In a similar vein to patterns, Petke [152] proposes the use of source code templates in genetic operators. These templates are proposed to be mined from software repositories, and could entail software fragments, program conditionals or variable ranges. Upcoming work by Callan et al. [153] expands upon this proposal by mining NFP related commits and categorizing them by the property being improved according to the commit message, such as run-time performance, network use or memory consumption.

Comparable to the suggestion of templates is the suggestion by Cody-Kenny, Fenton, and O'Neill [154] to re-use information identified during the GI search process and to use it as bounding information. The authors suggest doing this via data mining, treating the information recorded during runs as *big data*. Unfortunately, no further research by the authors in that vein beyond the position paper is known. KGGI (Section 3.4) works similarly, restricting the search space by upper bounds, and storing the entire search space for later mining of patterns. The upper bounds on requirements currently are only restricted via approximation that has been taken from performance measures (see Chapter 10), and does not utilize big data analysis as Cody-Kenny, Fenton, and O'Neill [154] suggest.

Krawiec and Swan [155] suggest a different use of patterns. Instead of attempting to identify recurring substructures in the source code, the authors attempt to identify recurring patterns in the data input and output. From this data, they replace the traditional fitness function with a decision tree that guides the fitness function. While our work does not deal with patterns in the same way as Krawiec and Swan [155], it may be interesting future work to identify the relation of AST patterns to input and output patterns.

7.3 Code Optimization for Compilers and Interpreters

There is a wide variety of optimizations available in compilers and interpreters [4], and a number of publications on the optimizations happening in the Truffle language and Graal compiler [61, 62]. This section concentrates primarily on optimizations that are similar in nature to this work. This concerns using patterns to modify source code for NFP, or automated bug fixing, only at the compiler level, as patterns themselves were discussed in Section 7.1. Techniques which may seem similar to GI, such as superoptimization [156] or code motion [13, 157] are also discussed. Explicitly not discussed are methods that use machine learning to improve existing compiler optimizations, such as using GP to create heuristics for optimizations [158–160] or unroll factor prediction [161].

Application of Patterns in Compiler Optimization

LIFT: `https://www.lift-project.org/`

A notable compiler that utilizes patterns is LIFT $^{\text{LIFT}}$. LIFT defines a functional programming language which uses rewrite patterns to deploy source code to multicore CPUs and GPUs, and utilizes hardware-specific optimizations. Since LIFT aims primarily to optimize parallelizable code, the applied rewrite rules concentrate on transforming the high-level source code into the low-level code executed on specific hardware. These rewrite rules are kept generic, so they can be applied in different sequences to achieve speedup by rule reordering. Some rewrite rules are specific to a target platform, such as rules specialized for OpenCL or field-programmable gate array (FPGA), achieving even more throughput than the more generalized rules [162]. LIFT itself is based on an AST which consists of expressions and lambdas. The rewrite rules transform these nodes to different nodes, manipulating the AST structure.

Recently, the authors of LIFT began to apply Deep Neural Networks (DNNs) to generate rule pipelines. These pipelines are a sequence of rules which are used to optimize source code for a specific hardware [163]. LIFT is not the only compiler which applies patterns on AST representations. Other examples are DELITE, which utilizes patterns to transform an intermediate representation (IR) of the source code to a specific CPU or GPU with a focus on parallelization [164]. Another example is Halide, which focuses on image processing pipelines. These pipelines are defined as algorithmic patterns, which are then chained together into a pipeline [165]. PADS is another compiler specialized for stencil operations often used in image computation. PADS optimizes source code with a pattern matcher in the parser and applies tuning to matched patterns [166].

All of the above examples are from the domain of functional programming. Pattern matching and rewriting is used outside the scope of functional programming, but this work mainly discusses design patterns [167]. One application of pattern matching, also with the goal of applying rewrite rules to source code in object-oriented programming languages, is TOM [168]. TOM is a domain-specific language (DSL) written in Java, and is used to match source code patterns which are rewritten via rules defined in TOM. The work does not utilize the compiler, however, and can be more considered a preprocessor for source code.

One other noteworthy compiler, which does not utilize patterns, is MilepostGCC [169, 170], which is a compiler designed for research purposes. The primary goal is to create a modular compiler, which is self-optimizing when ported to a new hardware platform. The compiler adjusts its optimizations for a specified set of targets such as execution time, compilation time or code size. While Milepost GCC does not consider patterns, it features a compilation interface integrating with the compilation steps, which enables making compiler internal information available to plugins. This is similar to how our work uses the information provided by Truffle such as the stack and heap information. Milepost GCC also utilizes a similar approach to a Knowledge Base as used by Amaru [171], by collecting compilation data, and storing it together with optimization heuristics in a database to continuously learn and improve on the compilation process. Recently, Fursin et al. [172] have extended their concept of a compilation database into the Collective Knowledge Framework (CK). CK provides a format and a database for machine learning workflows, again with a focus on automatically tuning multiple objectives in the non-functional domain.

Kartsaklis et al. [173] define HERCULES, a pattern-driven code transformation system. While HERCULES is not a compiler per se, it integrates with different compilers via compiler plugins. Hercules identifies human-written patterns in source code, and applies a human written rewrite to the identified pattern. HERCULES patterns deal with loop patterns, achieving a speedup of up to 67% [174]. The primary difference between HERCULES and the work presented here is that Hercules considers textual patterns instead of ASTs. Additionally, HERCULES does not attempt to automatically identify patterns, instead it only deals with manually identified and designed rewrite patterns.

Superoptimization

Superoptimization is an optimization technique comparable to the more recent work in the GI domain [112]. Superoptimization is also comparable to code motion [13]. All three of these methods basically attempt to re-order source code to still be valid, but improve run-time performance. Code motion does so systematically, often with manually written rules, whereas superoptimization and work in the GI domain attempt it via search-based methods. GI does so in the context of SBSE. The original concept of superoptimization does so by brute force search of a finite (sub)-set of a processor's instruction set to find the shortest program solving a loop-free set of instructions [156].

As the original exhaustive search approach was a severe bottleneck for superoptimization, more recent work does not perform a complete search anymore. Instead, conditions [175] or pruning [176] are applied to filter out invalid candidates before evaluation. Another approach to manage the search space is the application of constraint solvers, or satisfiability modulo theory (SMT) solvers [177].

Mukherjee et al. [178] present an approach to reduce the search space by pruning invalid solution candidates using data flow pruning. This is similar to one anti-pattern discovered during the experiment set (see Section 6.3). Both essentially prune the search space, excluding solutions which will result in an incorrect data flow. The primary difference is that [178] manually defines rules for checking, while our work applies a rule set discovered during a mining process. However, the pruning process of [178] conducts more extensive pruning, as it was designed by experts.

Superoptimization and GI gained more overlap in recent years. Research began to apply stochastic search on loop-free superoptimization. Schkufza, Sharma, and Aiken [179] define superoptimization as a multi-objective optimization problem and apply Markov Chains with a fitness function applied to the search space instead of completely evaluating it, showing a significant speedup in the optimization process itself while achieving comparable run-time performance with the resulting programs. Stochastic Superoptimization has also been successfully applied with loops [180], achieving a 25% speedup, by optimizing loop kernels.

Bunel et al. [181] consider superoptimization in the context of machine learning. They apply it differently to [179], instead opting for a sampling approach, to find out, which superoptimization steps are most likely to have the greatest impact on the run-time performance. On a learned Neural Network, they then predict the most likely candidate and apply the rewrite most likely to produce the best result.

Shypula et al. [182] show an alternative, machine learning approach, outperforming both SMT and neural networks. Their approach, called Self Imitation learning for Optimization (SILO), applies the Self Imitation learning (SIL) algorithm [183] on superoptimization. This is a reinforcement learning approach, where a ranking is applied to previous decisions made by the algorithm. The algorithm reinforces good decisions by applying a positive value corresponding to the observed outcome, but no negative values on failures. The approach is more generalizable to real-world problems than SMT constraint solvers and does not require a learning phase as neural networks do.

Especially when considering linear GP [184, 185], which is also applied to assembly code, and modifies the source code similarly as superoptimization does, there seems to be little difference between the two approaches. The two primary differences seem to be that GI considers bug fixing in addition to NFP and that there is a difference which algorithms are used. While both domains are firmly in SBSE, GI tends to apply evolutionary algorithms or local search methods, while superoptimization prefers neural network or reinforcement learning approaches. Additionally, GI considers multiple different representation forms outside linear GP, such as an AST or the source code.

Many of the challenges addressed in this our are similar to superoptimization. Both must deal with a large search space, and ultimately attempt code rewriting to improve NFP. Our work applies ASTs as a representation form, and also manipulates an AST in a pattern-based way before it is passed to the compiler. Superoptimization is generally applied after the compiler has already generated assembly code. Superoptimization can be considered an addition to usual compiler optimizations, just like our work.

Code Motion

Code motion is a method that looks similar to superoptimization on the surface. Code motion is the transformation of source code by only moving the placement of statements. This may include moving statements outside a loop, or simply reordering statements within a control flow block. Code motion is usually based on a set of rules, restricting the search space of acceptable motions to only allow movements that will not modify the semantics of the code. This results in a tradeoff between the cost for the analysis, and optimization and the effects that the motion will have on the program's run time [13, 157].

As register use is an important consideration where code motion can be applied, much research is put into this work. Banerjee et al. [186] discuss the application of finite state machines to apply code motion to statement blocks, successfully considering code motion across loops.

The primary differences between code motion and GI, are that code motion does not manipulate the code beyond moving statements, expressions or access-patterns, and that GI does not apply much validity checking. In the context of our work, code motion might be applicable in conjunction with the presented methods in GI, as the same verification mechanisms could be applied in the mutation statement.

Application of Machine Learning in Compilers

There are two recent surveys on the application of machine learning in compilers [187, 188].

Wang and O'Boyle [187] analyze machine learning applications in compilers in general, primarily identifying work that is applied to compiler autotuning. Autotuning is the concept of a compiler selecting their own optimizations, and deciding on the execution order of the selected optimizations. This is a challenging problem in compilation, as the multitude of available optimizations and different hardware platforms to optimize for, presents a challenging search space. Wang and O'Boyle [187] identify two core issues for machine learning in the context of compilers. One is the large run-time overhead, which makes it challenging to apply during the compilation phase itself. Another issue is the difficulty of gathering sufficient data, and the quality of said data. When the data has been made available, the application of machine learning can outperform regular optimizations [189, 190]. This work addresses the issue of data gathering by building a knowledge base, on which patterns are identified and verified. The problem of a large run-time overhead remains yet to be solved. While the patterns are applicable in KGGI, further work must be done to make them useable during compilation.

Ashouri et al. [188] present a similar view on the topic, with a survey specifically on the topic of compiler autotuning. They analyze more than 200 publications in the domain over the past 25 years, and show that primarily supervised, unsupervised and reinforcement learning approaches are applied in this domain. The publication also summarizes the efforts in the domain of genetic algorithms applied to autotuning. Concerning pattern mining or its application, they do not identify any work in addition to what has already been discussed.

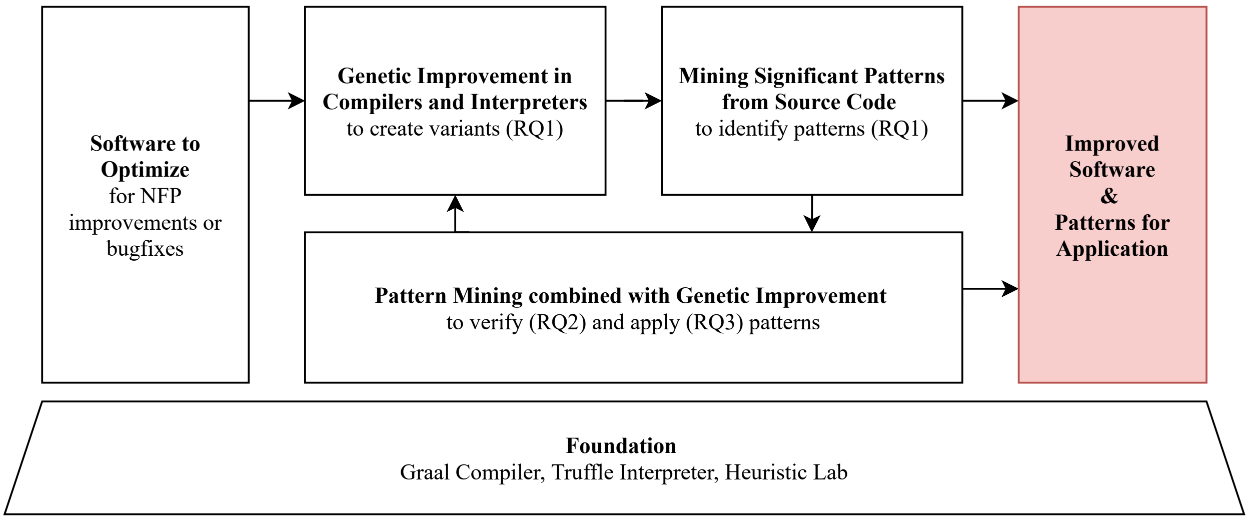
Machine learning is also applied to the Graal compiler, which is used by our work. Mosaner [191] describes the use of machine learning for deciding which optimizations should be applied in what order. Heuristics in this domain are primarily hand-crafted, and adoption of machine learning in this domain is low, as the approaches applied are primarily black-box machine learning. Mosaner proposes using data gathered on optimizations in a feedback-oriented machine learning approach that is then incrementally included as new heuristics in the compiler. As a first proof, this methodology is applied in code duplication by citeauthmosanerUsingMachineLearning2021. In their work, the authors apply machine learning to provide a trade-off function between code size increase and performance increase. They predict the code size increase for each optimization phase in the compiler by using an artificial neural network, gaining a code size reduction of 10% over several benchmarks.

Conclusion 8

The goal of this thesis is the identification and verification of patterns in software that influence Non-Functional Property (NFP). This is not the first work of its kind. Previous work has dealt with good design patterns to improve maintainability, quality or security [42, 43, 50], or to mine defects to either identify their location or fix them [37, 46, 49]. The work presented here conducts mining at the low level of an interpreter and compiler. This can produce very fine-granular patterns that can be applied to improve source code. This is possible in part because full access to the execution environment allows gathering more data (Chapter 4). Both mutational bug-patterns, and performance patterns and anti-patterns have been successfully identified using Independent Growth of Ordered Relationships (IGOR), a novel algorithm defined in this thesis.

These patterns can be applied semi-automatically to source code and positively influence Search-Based Software Engineering (SBSE), specifically Genetic Improvement (GI). This requires manual work to turn mined patterns into generally applicable patterns, or to identify interesting source code locations where patterns can be applied. The presented mining approach helps identifying patterns and introduces the novel concepts of generalization via taxonomies and wildcards, to not only make the patterns more expressive but to enable applying them to existing Abstract Syntax Trees (ASTs) (Chapter 5). In the future, this can enable not only the introduction of general optimizations, but automated refactorings and code maintenance, e.g., via automatically injecting defensive programming statements, logging, and rewriting code among others.

Knowledge-guided Genetic Improvement (KGGI) is a novel approach to GI. It uses a syntax graph which enhances a known grammar of a programming language with NFP knowledge to restrict the search space, and with requirements that must be met during the creation of an AST


Software to Optimize
for NFP improvements or bugfixes
Genetic Improvement in Compilers and Interpreters
to create variants (RQ1)
Mining Significant Patterns from Source Code
to identify patterns (RQ1)
Pattern Mining combined with Genetic Improvement
to verify (RQ2) and apply (RQ3) patterns
Improved Software & Patterns for Application
Foundation
Graal Compiler, Truffle Interpreter, Heuristic Lab

to expand it from syntactically valid, to semantically feasible. This, in addition to novel applications of test suites in the fitness function, has helped to produce mutants for the pattern mining approach in this work (Chapter 3). The patterns have also been shown to improve the GI approach via being injected into the syntax graph of KGGI helping to drive the search space into a positive direction and to prevent anti-patterns that lead to bugs. This doubles the individuals in the population and increases the amount of executable ASTs to 60.8% over only 20% in related work (Chapter 5).

In the following, answers to the initially stated research questions are given and possible future research directions are identified. This concludes the first part of this thesis, summarizing the novel contributions, concepts and algorithms. Part 2 of the thesis summarizes the frameworks implementing this work, and provides case studies to empirically evaluate it.

Research Question Summary

RQ1 How can recurring patterns be identified that impact or improve a functional or Non-Functional Property?

Patterns can be identified via the application of *Cluster Pattern Mining*, an extension of discriminative pattern mining, that mines a search space of AST that is grouped into multiple clusters that are discriminative in one NFP or functional property. Cluster pattern mining furthermore employs a *taxonomy* that represents the programming language the ASTs are written in. For each programming language, multiple taxonomies can exist with a specific focus, such as a focus on data types, or a focus on data flow. Mining includes *wildcards* which are added to patterns via *co-located pattern mining*. This is a process that first identifies patterns, then co-located patterns in the same cluster, and finallly outliers in other clusters.

Wildcards support the definition of node orders or indirect parent-child relationships via the (★) any wildcard. The (.) any node wildcard enables structural patterns without a specified context. The (¬) negation wildcard allows enforcement of faults of omission, or alternatively preventing nodes that should not occur in positive patterns. This approach is supported by the novel IGOR algorithm. It implements the cluster pattern mining approach and the mining of independent as well as embedded patterns (Chapter 4).

Our approach led to the successful identification of several mutational bug anti-patterns (Section 6.3), as well as patterns and anti-patterns influencing the run-time performance of software (Section 6.5).

The mining itself is done via the application of KGGI to produce different variants of code. Test suites are applied to verify the semantics of code, while aiming to produce multiple ASTs with different NFPs. Applying this in the context of a compiler or interpreter has the advantage that, via instrumentation and access to the execution environment (stack, heap, functions, ...), the NFP can be evaluated with a high precision and confidence. Additionally, as the mining is done on the internal program representation (AST) used by the compiler or interpreter, these patterns show a fine granularity (Chapter 3).

RQ2 How can the confidence in patterns be improved?

Improving the confidence in a given pattern is part of the process of *co-located pattern mining*. Its final step is the verification of patterns.

The verification of patterns is done by exclusively mutating ASTs to either include an anti-pattern (Section 6.3), or by identifying ASTs that contain an anti-pattern, and then applying a rewrite to conform to a pattern fixing the issues of the anti-pattern (Section 6.5). A confidence score is built from multiple mutations, where the confidence is measured in percent of the experiments that had the expected result. In this work, anti-patterns in the domain of mutational bugs in GI were successfully validated with an average confidence of 90.1%, meaning that when an anti-pattern was injected, the expected bug occurred 90.1% of the time. Corresponding fixes were validated with a confidence of 94%.

As the approach of using AST mutants applies patterns randomly in the AST this can also help to identify how a pattern can be improved or if there are limitations to the pattern. For example, in the conducted pattern verification (see Section 6.3), the verification phase showed additional conditions when a pattern identified from mining becomes applicable, enabling the refinement of the pattern.

A limitation of this type of verification, is that the confidence is negatively influenced by dead code. If the pattern is injected in a part of the AST that is never executed, this negatively affects the confidence score.

RQ3 How can these patterns be utilized to lead to general optimizations?

General optimization patterns which could be applied on their own, directly in a compiler or interpreter were not identified as of yet. The patterns that were identified, are mostly useful to direct search in GI.

To utilize patterns in GI the novel algorithm KGGI provides a syntax graph that manages the large search spaces that are the result of an application in the compiler or interpreter, as programming languages consist of hundreds or even thousands of concepts that can occur in a taxonomy. The syntax graph handles upper limits on NFP being produced and restricts anti-patterns that prevent the creation of unviable solution candidates. Alternatively, patterns can guide the search space into better directions. This produces more viable individuals that can be utilized in pattern mining.

In this work, 22 out of 25 AST were successfully improved by KGGI. On average, the run-time performance was improved by 39.2% for a selection of *math*, *sort* and *neural network* algorithms. Several optimization patterns have been identified, which provide a useful basis for further improving software via GI.

Future Work

The three main research questions of this work have been successfully answered. This lays a solid foundation for identifying and verifying NFP patterns. Future work can do still more, in many of the areas of this field. Throughout the work, it has been discussed why our approach has been chosen, and why other options have not been explored. Some of these are interesting for future work, starting out with the very foundation of the approach, the compiler and interpreter selected, as well as the language. This work is based on MiniC (Chapter 9) as a language for empiric validation. MiniC is written in Truffle and compiled via Graal. . Future work can look at many more compilers, interpreters, and programming languages.

A closer goal would be the analysis of Graal's intermediate representation (IR), either instead or in addition to the Truffle AST as a representation for pattern mining. This has not been approached in this work, as both the KGGI and IGOR algorithms depend on the representation form to be acyclic. This is not only interesting for the IR, but generally for other representations of software that may require a different mining approach, but can still benefit from the concepts of taxonomies and wildcards. Though it should be noted, that IGOR, can mine any tree representation not just Truffle ASTs.

The confidence in patterns (RQ2) could be improved even further. This work assumes that the given test suite of an AST is acceptable for the intent of the experiment. Future work could benefit greatly from utilizing an approach that synthesizes tests either to test ASTs that have no test suite, or even to automatically adapt the test suites to improve the coverage of modified ASTs. Similarly the pattern verification has been shown to become less conclusive if the patterns are introduced into dead branches. Extending the approach to only consider branches that are executed could improve confidence in patterns. This also holds for the mining approach itself. Dynamic information could be rather beneficial. Dead branches could be removed from mining, reducing false positives and making the code actually responsible for NFP more discriminative. The metrics could also benefit from additional information, such as how often a loop is executed or how likely a branch will be taken.

In the approach of *cluster pattern mining* the creation of clusters could possibly be automated, via clustering methods from literature, for example via the function signatures (input and output) of AST in the functional domain or via the NFPs.

The concept of wildcards in patterns could be improved even further in the future. Throughout the work, it has been shown that in some cases the patterns could benefit from having more expressiveness. One example being a wildcard that could allow a pattern or a part of it to become independent of the node order. This would not just benefit changes in the AST but also the identification of anti-patterns, which can occur in multiple orderings. For example, in which order the child nodes of an *and* comparison are grouped may not always be relevant. Especially for fine-granular patterns, commutative and associative properties may be explored in patterns.

In the area of KGGI and the dynamic fitness functions, some future work is the introduction of additional operators that specifically tackle how tests are applied. Similarly diversity metrics could drive the search both in KGGI and in mining locations where patterns could be applied or found. The syntax graph of KGGI, as a graph and selection mechanism in which AST nodes will be created, has similarities to a ConvolutionalNeuralNetwork. Following this concept in the future may be an interesting research direction as well.

The framework Amaru (see Chapter 10) and all other frameworks worked on in this thesis are open source. Now that the foundation has been set, it would be great to see it be used by the community. Additional ideas and expertise from other researchers can greatly benefit this work. Much of the future work will include case studies and showcases of how pattern mining and GI can greatly benefit from the application directly in an interpreter or compiler.

Frameworks and Case Studies

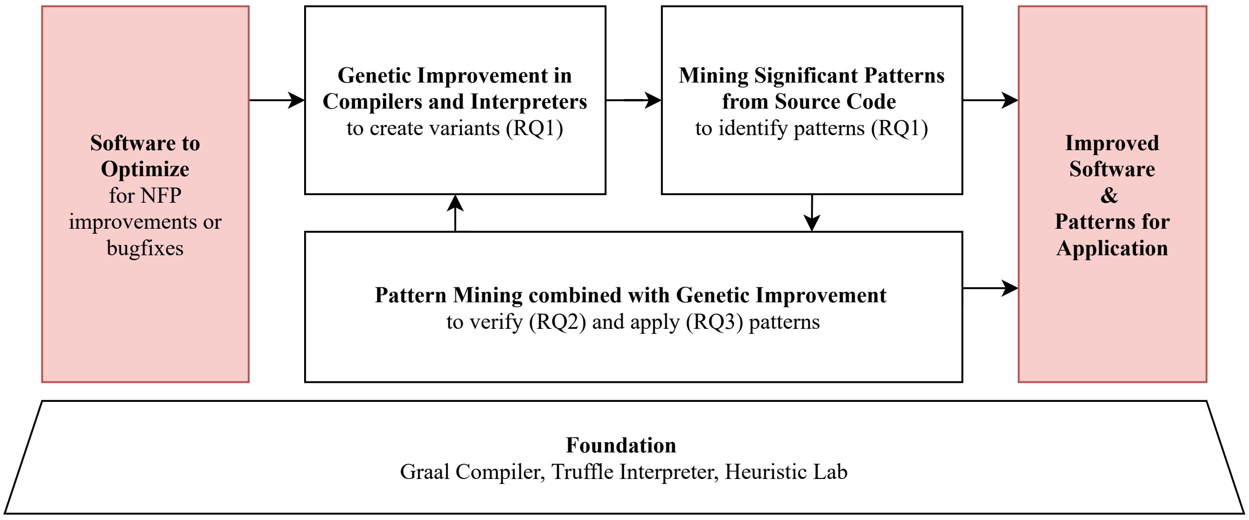
MiniC | 9

MiniC is a language developed via the Truffle framework [60] for testing our approaches in the Genetic Improvement (GI) and pattern mining domains. It is a subset of the C11 version of C [84], and consists of some core mechanisms such as function declarations, loops, constants and variables (both global and local). The Extended Backus-Naur Form (EBNF) grammar is provided in Listing 9.1. MiniC does not support all data types of C, and also does not support pointers or structs. There are two deviations from C11:

- Arrays are valid function parameters. Since MiniC does not yet support pointers, this had to be added to the language, to enable more interesting test cases.
- Different exceptions. Due to the advantages of Truffle, some undefined behavior of C will lead to different outcomes than in regular C compilers. For example, access outside array bounds will lead to an Array Out Of Bounds exception, which in C would either not lead to an exception or to a Buffer Overflow error.

Truffle itself is a framework that deals only with the definition and interpretation of a given language, and does not provide a parser. The parser has been implemented via the Coco/R framework [192, 193].

At the time of writing, MiniC consists of 362 real or abstract node types, of which 167 are instantiable and can be used as operators and operands in GI. Of these, 59 classes are terminal classes. The primary difference to non-terminal classes being that they do not contain relationships to other MiniC classes and thus can only be leaf nodes in an Abstract Syntax Tree (AST). The reason why there are many abstract classes is that Truffle nodes are implemented as Java classes. Some of these represent abstract functionality, such as a core MiniC node, from which all other nodes in the language inherit directly or indirectly. Many abstract nodes are automatically generated by Truffle from the given implementations,


Software to Optimize for NFP improvements or bugfixes
Genetic Improvement in Compilers and Interpreters to create variants (RQ1)
Mining Significant Patterns from Source Code to identify patterns (RQ1)
Improved Software & Patterns for Application
Pattern Mining combined with Genetic Improvement to verify (RQ2) and apply (RQ3) patterns
Foundation Graal Compiler, Truffle Interpreter, Heuristic Lab

which is explained further in Section 2.4. While the size of MiniC already is a challenge for the search spaces in GI, it is still more manageable compared to fully implemented languages. For example, the JavaScript[1] implementation of Truffle, consists of 854 instantiable node types.

1: https://github.com/oracle/graaljs

The implementation with Truffle provides several advantages concerning the concepts of Knowledge-guided Genetic Improvement (KGGI) (see Section 3.4) and the utilization of hierarchies during the pattern mining process (see Section 4.2). Due to how Truffle languages are implemented, a natural hierarchy already exists in the implemented class hierarchy. This is influenced by the language's core concepts, shared implementations, and data types. In MiniC, the focus lies on the different data types, as the nodes are grouped by their return type.

Listing 9.1: EBNF of the MiniC Language

```
Minic = Program.
Program = {(VarDecl | ProcDecl | ConstDecl StructDecl<out
    MinicNode struct>)}.

Type = ident.

ConstDecl = "const" Type ident "=" (intCon | floatCon |
    charCon | stringCon)";".
VarDecl = Type VarOrArrayDecl {',' VarOrArrayDecl} ';'.
VarOrArrayDecl = ident {'[' Condition ']'}.
ProcDecl = (Type | "void") ident "("[FormPars]")"
    (BlockStatement | ';').
FormPars = Type VarOrArrayDecl {"," Type VarOrArrayDecl}.

Statement = (ConstDecl | StructDecl | ReturnStatement |
    BlockStatement | IfStatement | WhileStatement |
    EmptyStatement | Designator ("=" Condition | ActPars)
    ";") | VarDecl).
WhileStatement = "while" "(" Condition ")" Statement.
ReturnStatement = "return" Condition ";".
IfStatement = "if" "(" Condition ")" [ "else" Statement ].
Condition = ("!"Condition | CondTerm { "||" CondTerm }).
CondTerm = CondFact {"&&" CondFact}.
CondFact = Expr [ Relop Expr].
Designator = ident {("." ident ) | ("[" Condition "]")}.
ActPars = "(" [Condition {"," Condition}] ")".

Expr = Term {Addop Term}.
Term = Factor {Mulop Factor}.
Factor = (Designator [ActPars] | intCon | floatCon |
    charCon | stringCon | Addop Factor | "(" (Type ")"
    Factor | Condition ")")).

EmptyStatement = ";".
BlockStatement = "{" {Statement} "}".
```

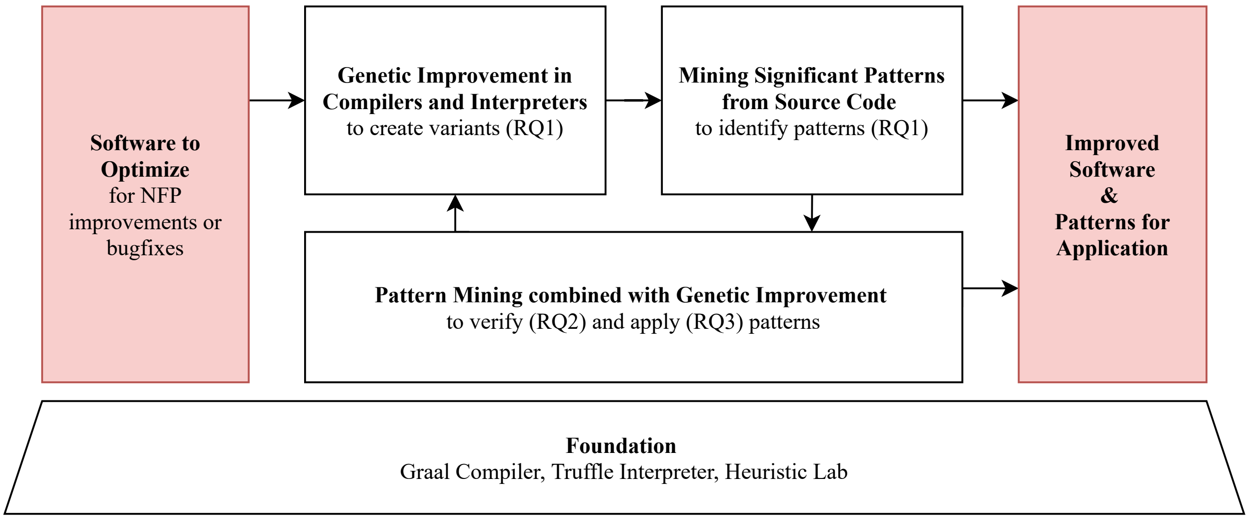
Amaru | 10

Amaru is the core framework that was developed in this thesis. It contains reference implementations of the presented concepts and algorithms in our approach. This includes the enhancements of Genetic Improvement (GI) such as Knowledge-guided Genetic Improvement (KGGI) (Section 3.4), as well as our pattern mining approach with the Independent Growth of Ordered Relationships (IGOR) algorithm (see Algorithm 6). The concepts concerning co-located pattern mining (Section 4.7) and pattern verification (Section 5.3) are also implemented in Amaru. It is implemented in Java 11 on the Graal VM (version 21.1.0), utilizing Truffle. Amaru itself is primarily a framework and does not come with a user interface. In the domain of pattern mining, visualization is necessary. For this Amaru provides a reporting service to view results in different formats such as Markdown, Hyper Text Markup Language (HTML), or LaTeXdocuments.

Open Source - Reproducible Experiments

Amaru is available open source under the Mozilla Public License 2.0, a permissive license that was chosen to enable others to use the technical contributions made here for their own scientific work. [171] is a registered DOI to the exact version of Amaru that the experiments were conducted with. Guidance on reproducing these experiments is given in the repository.

[171]: Krauss (2021), *Amaru - The Amaru Framework for Genetic Improvement and Pattern Mining in Graal and Truffle* https://doi.org/10.5281/zenc


Software to Optimize for NFP improvements or bugfixes
Genetic Improvement in Compilers and Interpreters to create variants (RQ1)
Mining Significant Patterns from Source Code to identify patterns (RQ1)
Improved Software & Patterns for Application
Pattern Mining combined with Genetic Improvement to verify (RQ2) and apply (RQ3) patterns
Foundation Graal Compiler, Truffle Interpreter, Heuristic Lab

10.1 Architecture

The architecture of Amaru, (see Figure 10.1) was previously presented in [65]. It builds on top of the Truffle API, and is designed to be executed directly in the Graal VM. The functionality of the framework is split into two parts. The optimization part of the framework deals with the application of GI, directly in Truffle languages. The pattern mining part of the framework also utilizes information from the Truffle languages and can inject source code transformations in a guest application. Both parts of the framework access a knowledge base storing information on the Truffle *guest language*, and experiments conducted in the language.

The optimization side of the framework allows the creation of *experiments*. An experiment consists of the program that should be run in a Truffle *guest language*, such as MiniC, the function that should be optimized, a test suite to verify that function, and a configuration. The configuration contains, for example, the fitness function that has been selected, and the settings of the *optimizer* that will be used during the *experiment*.

The *optimizer* has multiple implementations. Aside from a reference implementation of KGGI, other genetic algorithms and operators are implemented. The optimizer enables a connection to other optimization frameworks, to enable Amaru being used as access to Truffle guest languages while still utilizing the algorithms and user interfaces provided by other optimization tools such as Heuristic Lab (see Chapter 11).

Amaru can be a valuable addition for other frameworks and can serve as an intermediary, as it also helps preparing Truffle languages for the

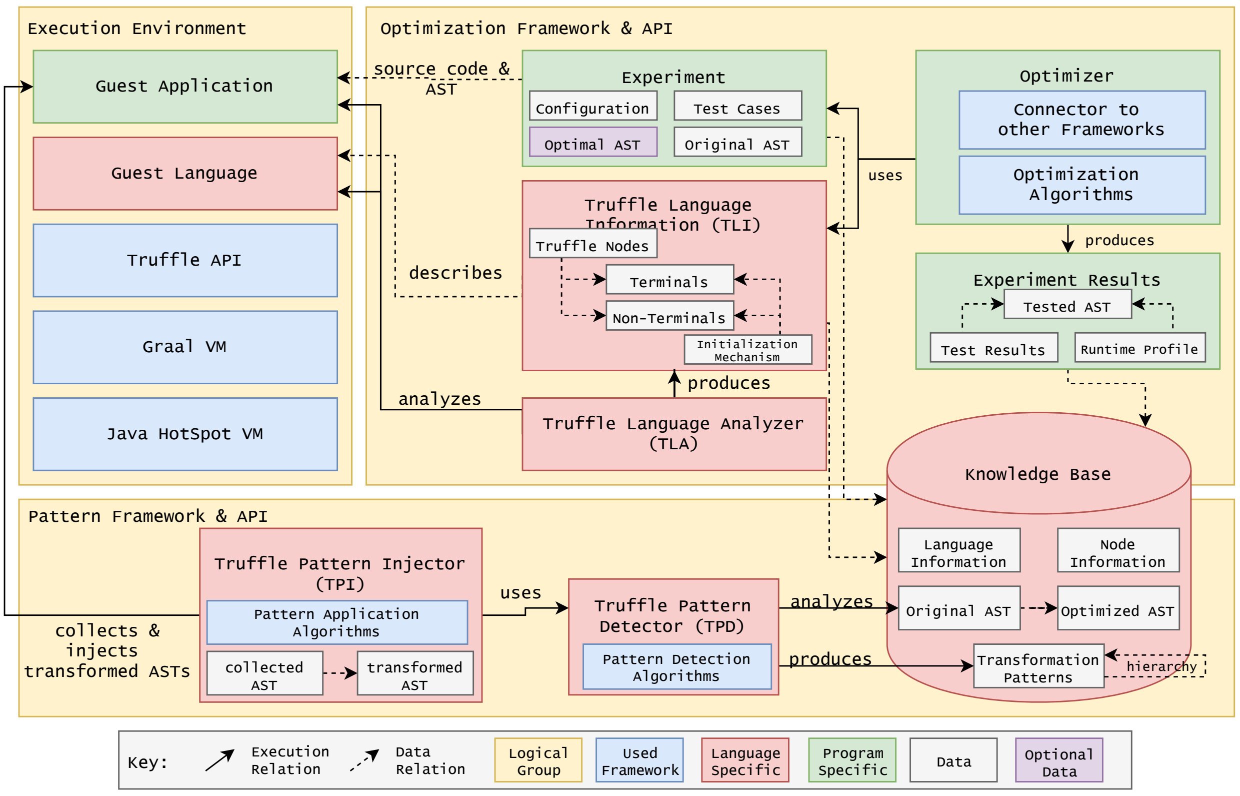


Figure 10.1: Architecture of the Amaru framework. It builds upon the Truffle and Graal execution environment, and consists of functionality for optimization using GI, and the IGOR algorithm to conduct pattern mining. The experiment data is stored in a knowledge base.

purpose of machine learning. This is because it automatically extracts necessary information for machine learning, and abstracts Truffle languages for that purpose. It also provides an evaluation methodology specific to the combination of genetic algorithms and Graal, preventing threats to the validity in the measurement of Non-Functional Properties (NFPs).

The *knowledge base*, a Neo4J graph database, serves to store information about a given Truffle Language, and about conducted experiments. For the experiments, this primarily concerns the Abstract Syntax Trees (ASTs) executed during an experiment and the respective results of that execution, e.g., if an error occurred, how close the test results are to the expected outcomes of the test suite, and the measured NFPs.

The pattern mining side of the framework can later load data from one or multiple *experiment*s, and can also load information about the utilized Truffle language. It contains a reference implementation of the IGOR mining algorithm, including a reporting tool to view the results of mining experiments, as well as functionality to rewrite Truffle ASTs according to a given pattern or to inject such a pattern in later GI experiments.

10.2 Language Analysis

In the context of GI or Genetic Programming (GP), Truffle languages are a good basis, as their nodes are automatically executable. However, additional information about a language is needed. This includes how nodes have to be instantiated, which nodes influence the control or data flow, and call other functions. Making this information available would be an unreasonable overhead for developers of a specific Truffle language and would produce a large overhead for integration in other frameworks, especially frameworks that are not written in Java.

Amaru deals with this by providing an automated analysis for any Truffle language. Amaru provides information and construction methods that are independent of specific Truffle languages. External frameworks can load information about languages available in Truffle and either utilize just the provided construction methods themselves or utilize parts of the implemented algorithms in Amaru, such as crossover or mutate operators that are already implemented, mixing them with their own algorithms and operators. Alternatively, the frameworks can use Amaru exclusively to evaluate ASTs according to a given set of tests and NFPs.

The information extracted from Truffle languages primarily concerns the available Truffle nodes and is gathered automatically via Java Reflection. The only information that has to be provided are the names of the packages in which the nodes are implemented in the given Truffle Language. The language information is provided in the *TruffleLanguageInformation* as seen in Figure 10.1. It contains all classes that are derived from the Truffle *Node* class that all nodes must implement in order to be executable in Truffle. These classes are represented as a hierarchy, where each class or interface points to each possible instantiable implementation. Each instantiable class is analyzed and the following information is collected:

Class name the fully qualifying class name of the implementation.

Metadata such as the short names and descriptions that developers of a language can optionally assign to their implementations.

Properties of a class, such as if the node influences the control or data flow.

Initializers of the given classes are recorded, including their parameters.

NFP approximations are derived by a learning process. This concerns the NFP costs of nodes, such as their run-time costs.

The recorded hierarchy, in conjunction with the initializers for each class, enable the generic approach of Amaru when dealing with Truffle languages. From this, the Syntax Graph from KGGI (see Section 3.4) can be generated to mutate and cross valid ASTs. When a node is a non-terminal and requires child nodes, they will either be pointing to an abstract class of which all known implementations are recorded in the *TruffleLanguage-Information*, or it will be pointing to a specific implementation already. Some parameters of a Truffle node are non-language nodes. Among these are literal values, variable names or function names. For each of these there must still be a manual implementation that provides values, though most cases are automatically handled by Amaru, providing default values for simple data types, and automatically recording the arguments of a function. Access to these functions has to be handled per language, as there is no dedicated function registry in Truffle.

The observed properties, also derived via reflection, record if a given node class has access to a FrameSlot, which in Truffle represents the stack or the heap. From a call analysis it can be derived if the access is writing or reading, and if it is to the stack or to the heap, respectively. Access to function arguments, i.e., mostly nodes transferring call arguments to the stack, is also recorded. Additionally, node classes that influence the control flow, such as branches and loops, are identified. Node classes conducting function calls are recorded as well.

In addition to the static approach via Java Reflection, a dynamic learning process is applied. This partially concerns valid pairings of allocations, reads and writes, such as which implementation of a writing variable access correctly initializes a stack or heap variable, so it can be read later by another node implementation. It also concerns which nodes are responsible for which data type when reading function arguments. And finally, the dynamic approach is responsible for assigning the NFP approximation values. Currently, this concerns only the run-time performance of every node.

For each node, it is measured individually, how much run-time it's execution costs before Graal has finished its warm-up phase, and after it has finished it. The exact details of the measurement are discussed in the next section. The dynamic learning process requires a manually written empty function written by a developer. This function is first executed 1,000,000 times to measure its empty run-time cost. After this, every instrumentable node in the language is injected into that function stub. It's executions are measured via *in-process iterations* to minimize side effects of measurements [81]. Every node is injected 10,000 times in the stub. This poses two challenges that need to be addressed. First, not every node can is terminal. This makes it challenging to record its run-time performance, as it will always be dependent on the nodes in its relationships. Second, Graal will optimize the node away if it is not used in a meaningful context, e.g., when its return value is not accessed.

For nodes that cannot be utilized on their own, an incremental growth approach is used. For example, a node requiring child nodes is only analyzed after it's required children have already been weighted. A trace analysis of a single execution of the AST is used to derive which nodes were actually executed how many times in case of branches and loops. The weight of all executed child nodes is subtracted from the final weight of the parent node. Similarly, nodes requiring other nodes to be executed before them, such as a reading variable access, have one single initializing node injected before them, with only the node currently being measured injected multiple times. For example a write node to variable x is injected once, and the read node to variable x is injected 10,000 times.

To prevent Graal from optimizing the nodes away, for example, if they have no impact on any return value, they would have to be measured in a meaningful context. For example, variable reads would be optimized away unless they were used in an assignment or a return statement. It is infeasible to construct meaningful variants for the several hundred node implementations of any given language and would skew the results, as individual nodes could never be measured without bleedover from other nodes. Thus, an interceptor is injected into every node class. The interceptor is annotated with a Truffle Boundary, which is a special compiler directive preventing optimizations from happening in them. This is also how the trace information is collected during the dynamic execution, on which nodes in a given AST are actually executed or contain the hot path via their execution count. While this prevents optimizations from happening, and thus threatens the validity of the measurements, it ensures that all nodes in the in-process iterations remain there for measurement. The cost of the intercept itself is not relevant to the node weight, as all nodes have that cost added to them equally.

The above way of measuring and assigning node weights comes with some caveats, especially since it is known that it influences the compiler to not conduct some optimizations. However, an exact weight could never be produced, as AST nodes influence each other, and larger sub-ASTs will be optimized by the compiler in different ways. This means that an exact prediction can never happen, unless every possible combination of nodes in ASTs would be measured with every possible compiler optimization setting, which represents an impossible search space size.

The approximated values are meant for the KGGI approach, to prevent the generation of infeasible mutants in the population. This is done by using the measured run-time performances of the nodes in a function that approximates the run-time performance of the AST instead of running the AST. The approximation is used as an upper limit, that must not be exceeded for the AST to be viable.

The function works by adding the measured weight of each node in the AST to a total sum via a breadth-first iteration of the AST. For branching statements, the weights of the sub-AST in the respective branch is reduced by an assertion which branch is more likely to be taken. For example in an *if* statement the *then* path is weighted with 0.7 and the *else* path is weighted with 0.3. The weight of a sub-AST in a loop is multiplied by an assertion of how often the loop will execute.

10.3 Accurate Measurement of Non-Functional Properties

The accurate measurement of a NFP, especially run-time performance, can be challenging in a modern compiler [81]. Challenges include the side effects from the garbage collector or other processes on modern operating systems. Differences in hardware, operating systems and compiler versions can make the measurements incomparable with other systems. Also, the compiler flags with whihc the code is compiled and executed change the run-time behavior.

One of the already discussed challenges is that in-process iterations of code, e.g., duplicating the code, so that it is executed multiple times in the same function, is not always feasible, due to the compiler possibly optimizing superfluous code away. The meaning of the code may also be changed due to being executed multiple times.

Another unique challenge when using Graal, but possibly also affecting other compilers, is *code caching* [57, 59]. This is less of an issue during benchmarking for compiler optimizations. The measurement can be made more accurate by considering the warm-up time of a compiler and omitting initial repetitions of a program execution from the measurement. However, in GI, due to the high cost of compiling and executing an entire code base instead of single functions, it is not unusual to only conduct 10 repetitions [112] of every individual in a GI population. This would already be an untrustworthy measurement, considering that the warm-up of Graal is considered to be 100,000 iterations. Additionally, in GI individuals in a population are often very similar due to grafting. The crossover and mutation operations only affect small parts of the source code as well, leaving much of the AST unchanged.

The combination of code caching, a small amount of repetitions per individual, and a large amount of similar individuals in a population over many generations make the perfect recipe for disaster when it comes to run-time performance measurements. The unusually similar code can allow the compiler to apply cached snippets over multiple individuals in the population. Thus, it may appear that the GI algorithm is producing increasingly faster performing individuals. What might really be happening, is, that the compiler continues to optimize the performance of a snippet that is executed increasingly often and still not fully profiled since the warm-up has not completed.

This also was an unknown issue in Amaru for some time, until, due to the pattern mining approach, virtually equivalent individuals were shown as outperforming each other. In some cases, for two recurring individuals, it was even observed over several experiments that one performed sometimes better than the other and sometimes worse. A closer look showed that the only real difference was which individual was executed in a later generation of the experiment.

Due to the presented challenges, when conducting GI experiments, Amaru utilizes several *executors*. An executor is responsible for running one single test case with one single AST. In the simplest case, e.g., a fitness function only having functional concerns, an internal executor can utilize Truffle for execution. Due to the instrumentation when learning a language, or tracing which nodes in an AST are executed, alternative executors exist that transfer an AST for testing into a different execution environment where the byte code of the Truffle guest language has been modified with instrumentation code. This is also done whenever the run-time performance of an AST has to be measured. A specialized executor starts an entirely separate Graal VM and transmits the AST to be executed together with the test case input and the program in which the AST is embedded. The AST is then executed in a completely separate environment. The executor waits and receives the output and 200,000 runtime-measurements for the execution, the first 100,000 representing the warm-up and the next 100,000 the observed performance measurements, the smallest of which is the peak performance. This external executor can also be stopped after a defined time to guarantee termination when it is not know if the AST will finish in finite time. This approach also encapsulates exceptions from which the Java Virtual Machine (JVM) cannot recover, such as when the garbage collector overhead limit is exceeded, or stack overflow exceptions. Otherwise, these issues could stop the entire experiment. This is an alternative way to solve this issue compared to Orlov and Sipper [26], who instead opted to instrument the source code with abort conditions when infinite loops or too many calls occurred.

10.4 Pattern Mining Reports

Pattern mining has been discussed in Chapter 4. It has been mentioned in Section 4.7 that dealing with redundancy and the accuracy of results actually depends on providing filtering and viewing methods to make the results easier to understand. This is done via *filtering*, *merging* and *relationships*.

The *filtering* can be done via AST constraints by excluding or including specific nodes or types of nodes. This can also be done via patterns, i.e., mining only in trees that contain a specific pattern or anti-pattern, which is relevant for the co-located pattern mining approach. *Filtering* also concerns automated ways in which the different clusters can be selected. This includes:

NFP selecting clusters by a range in one or more properties such as run time.

Functional properties such as a specific test input or output. For non-feasible individuals, this also allows selecting by the exception thrown.

Experiments ASTs from one or more experiments can be selected to analyze differences or similarities in different algorithms or domains.

Merging is primarily done via a containment approach. Patterns can be filtered by a *closed* approach, in which they are filtered if they are a complete subset, i.e., occurring in the same locations, as a larger pattern. This returns the largest unique patterns, but often produces many patterns around a similar core that is different only on the edges of the pattern. Alternatively, a *compact* approach does the exact inverse, filtering out all larger patterns that can be reduced to a smaller core. This

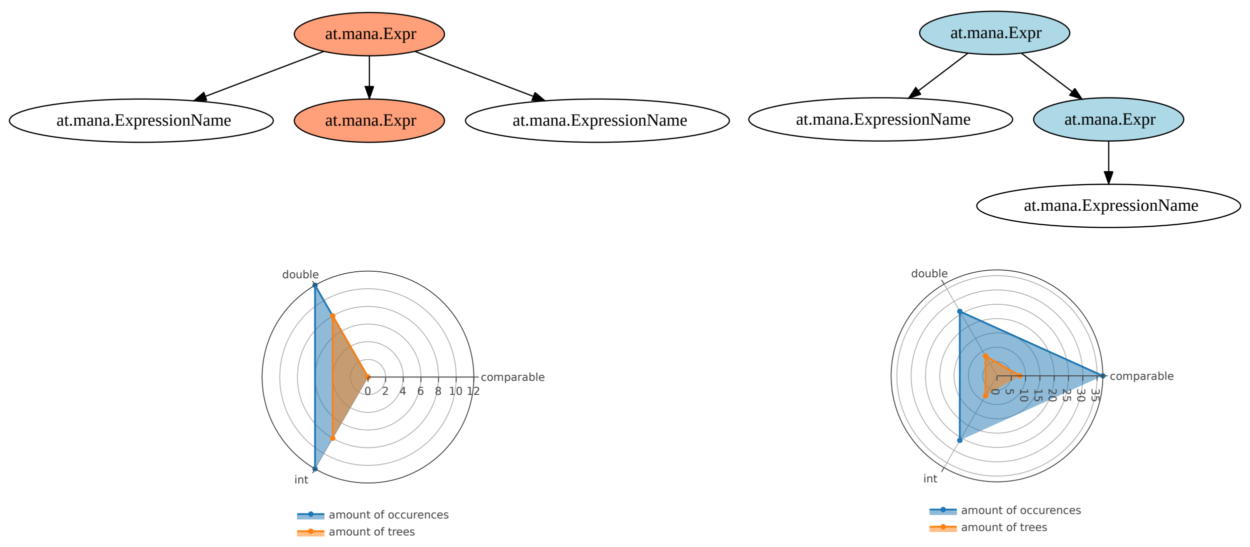


Figure 10.2: Visualization of two patterns that were mined with three clusters, int, double and comparable. Nodes with a red background are manually selected to identify nodes with a blue background in other patterns that are overlapping.

produces much fewer patterns than the maximizing approach and easily identifies pattern cores, though more detailed information can be lost.

Finally, the *relationships* can help to reduce patterns as well by showing only the most generalized or most specialized patterns.

A reporting UI was created that shows the results of a pattern mining experiment after the filtering and merging process has been conducted. The report contains visualizations of the search space, a visualization of all patterns found per cluster, or the differences per cluster. How the differences are visualized is shown in Figure 10.2. Each pattern is assigned a radar chart that shows how often it occurs in a given cluster. The Report is interactive and allows selecting nodes or patterns (red background), which shows the same node locations in all other patterns (blue background). This allows comparing overlap between patterns during analysis.

HeuristicLab Connector 11

The HeuristicLab Connector is a connector between *Amaru* and *HeuristicLab*, which is an optimization framework for meta heuristic and evolutionary algorithms. It provides a multitude of algorithms and reference problems from literature. HeuristicLab supports setting up experiments, and features a graphical user interface (GUI) that can be used to configure experiments and to visualize results. HeuristicLab already supports Genetic Programming (GP) via a domain-specific language (DSL) written and interpreted directly in HeuristicLab [194–197].

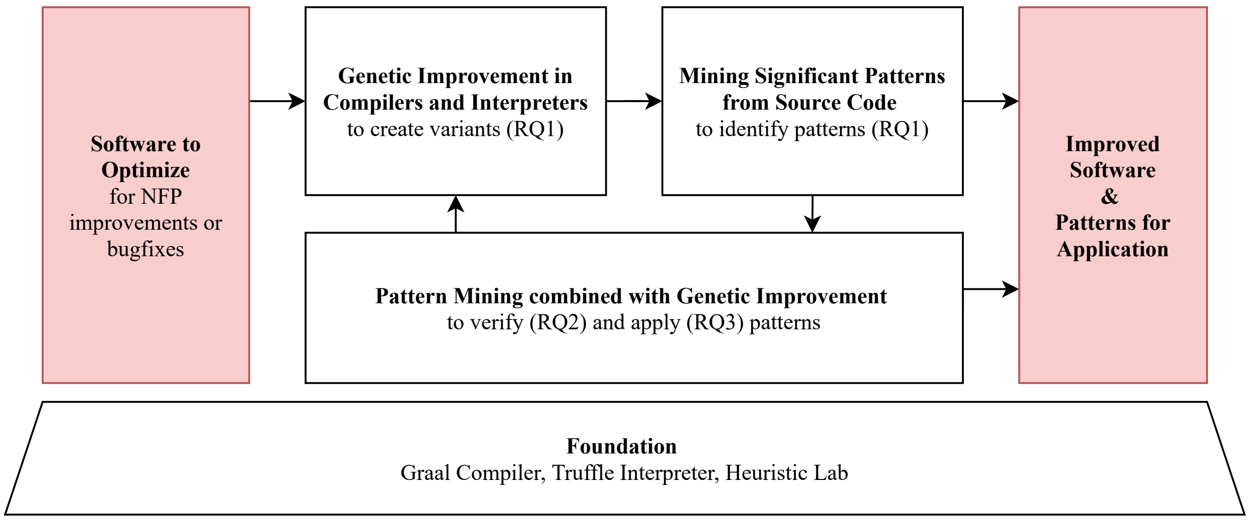
Open Source

HeuristicLabConnector is available open source under the Mozilla Public License 2.0. [198] is a registered DOI to the exact version of HeuristicLab Connector that this chapter describes.

[198]: Krauss et al. (), *HeuristicLab Connector - Connecting HeuristicLab to the Amaru Framework for Genetic Improvement and Pattern Mining* https://doi.org/10.5281/zenodo.6025063

The HeuristicLab Connector [199], is an intermediary to integrate a low-level execution environment, i.e., Amaru, with a high-level optimization framework, i.e., HeuristicLab. While the connector currently only integrates these two tools, it's core architecture, shown in Figure 11.1 is designed around a broker that accepts messages from HeuristicLab and forwards them to one or more instances of Amaru. The broker works with a message queue infrastructure, allowing either side to be replaced with a different framework.

The consideration of an integration of Amaru, and therefore the reference implementation this work is providing for its scientific contributions, lies primarily in the goal to enable others to utilize the work in their own research. An integration with HeuristicLab is beneficial, as this provides a GUI that allows easy definition of experiments, configuration of the algorithms provided by Amaru and visualization of the experiment execution and results. The integrated framework HeuristicLab benefits


Software to Optimize
for NFP improvements or bugfixes
Genetic Improvement in Compilers and Interpreters
to create variants (RQ1)
Mining Significant Patterns from Source Code
to identify patterns (RQ1)
Improved Software & Patterns for Application
Pattern Mining combined with Genetic Improvement
to verify (RQ2) and apply (RQ3) patterns
Foundation
Graal Compiler, Truffle Interpreter, Heuristic Lab

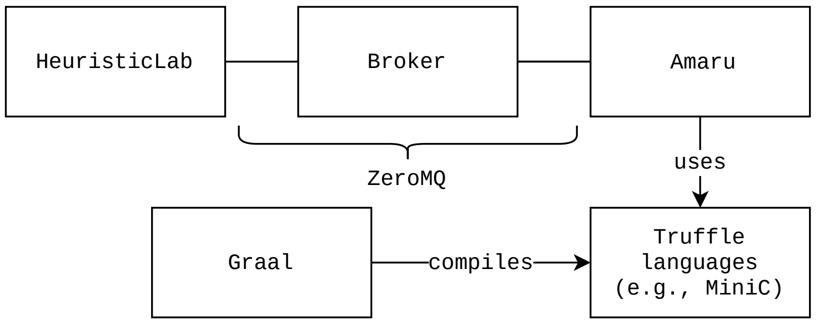


Figure 11.1: Overview of the technologies used in the HeuristicLabConnector (adapted from [199]).

from the direct integration with a compiler, i.e., Graal, and thus the ability to step beyond DSLs, enabling research into real-world programming languages and compilers.

The following sections explain HeuristicLab and considerations towards the infrastructure and messaging system of the Connector to Amaru. The core considerations of this approach are to provide an infrastructure that is independent of a specific operating system or framework to make them interchangeable, and enable distributed processing. As such, a broker can process multiple experiments at the same time, and can have workers on multiple machines process requests.

11.1 HeuristicLab

HeuristicLab is an extensible, modular framework for optimization problems featuring a wide variety of heuristic and evolutionary algorithms, as well as a GUI. HeuristicLab is written in C#. It already provides functionality to conduct evaluations with external software, for example the *HL3 External Evaluation Java library*[1] to integrate with Java applications. HeuristicLab also allows the extension of itself via it's plugin infrastructure. A plugin can add a new problem definition, algorithm, or operator for an algorithm to HeuristicLab. This is also utilized in the HeuristicLabConnector [194–196].

1: `https://dev.heuristiclab.com/trac.fcgi/wiki/Documentation/Howto/OptimizeExternalApplications`

What is not used in the HeuristicLabConnector is HeuristicLab Hive. Hive is a specialized software that HeuristicLab can communicate with, to conduct distributed experiments on client PCs that have Hive installed. The reason why it is not utilized is that Hive does not integrate well with external evaluations and that the HiveServer does not have a deeper understanding of the jobs, assuming that every client has the same functionality. This is not feasible for Graal, which may have different languages and versions of those languages installed on different PCs [200].

11.2 HeuristicLabConnector Architecture

The architecture of the HeuristicLabConnector is made up of a plugin for *HeuristicLab*, a *worker* extension for Amaru, and a *broker* connecting multiple HeuristicLab instances with *worker* instances via a Message Queue. All parts of this infrastructure are shown in Figure 11.2. The architecture asserts that the external framework HeuristicLab drives

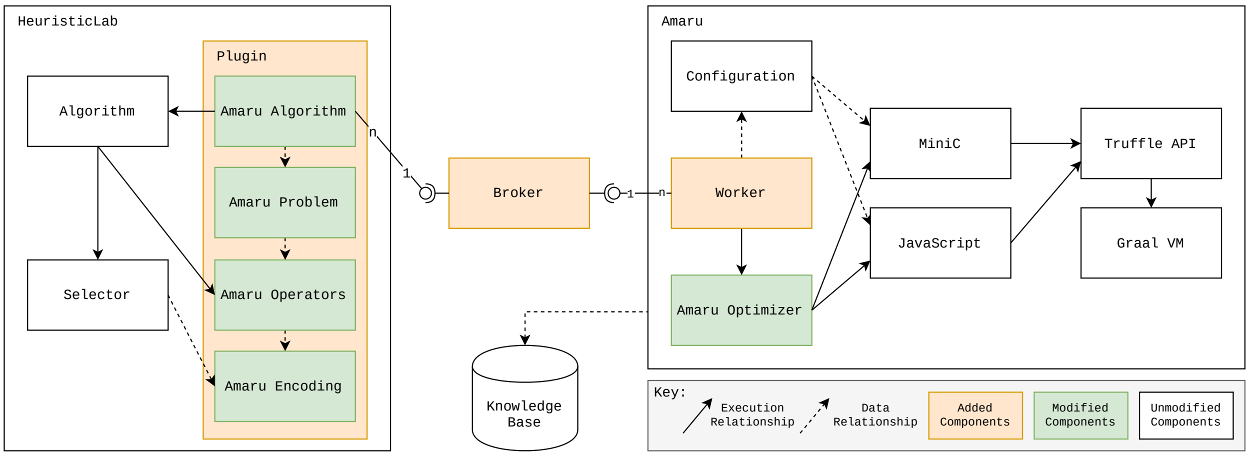


Figure 11.2: Architecture of the HeuristicLabConnector, showing the plugin that was added to HeuristicLab, the Worker that was added to Amaru, and the Broker connecting the two frameworks (from [199]).

the process. Thus, it is designed with a fine-granular operator concept, following the evolutionary operators, create, mutate, crossover, select, and evaluate. A request can be made to the broker with any of these operators, and a given Abstract Syntax Tree (AST) or multiple ASTs in the case of select or crossover. Thus, HeuristicLab has the option to drive the entire process and only use Amaru for evaluation, or it can opt to utilize the functionality of Knowledge-guided Genetic Improvement (KGGI), and its syntax graph to create, mutate or cross trees with a higher chance of them being feasible. As all workers still access the knowledge base and store generated or evaluated ASTs, there is no disadvantage to using the HeuristicLabConnector when it comes to a later pattern mining of the Experiment results.

The *HeuristicLab* plugin is based on the tree-based genetic programming functionality already existing in HeuristicLab [197]. The plugin contains an algorithm wrapping around any HeuristicLab algorithm. The wrapper handles the connection to the broker and injects the operators into the contained algorithm. The plugin also contains these operators, which can be selected from HeuristicLab operators, or connect to the broker to call the implementations Amaru provides. Finally, HeuristicLab contains the problem definition, which consists of test cases, and the source code to be executed on an Amaru worker as well as the necessary encoding of ASTs for transmission via the broker.

The *broker* is a lightweight message queue, that accepts connections from workers and stores their capabilities. When a request from HeuristicLab arrives, the broker knows via these capabilities which workers it can forward the requests to. The broker also handles load balancing by forwarding requests to unoccupied workers and manages failures via a heartbeat signal that lets it know if a worker has crashed and the request sent to it must be handled by another worker.

Amaru *workers* are an implementation in the Amaru framework, accessing and publishing the functionality of the Amaru optimizer (see Chapter 10). The workers also use the TruffleLanguageInformation to load installed languages and publish that information to HeuristicLab upon request.

The process of handling a Genetic Improvement (GI) experiment in this system begins with HeuristicLab loading the available operators it

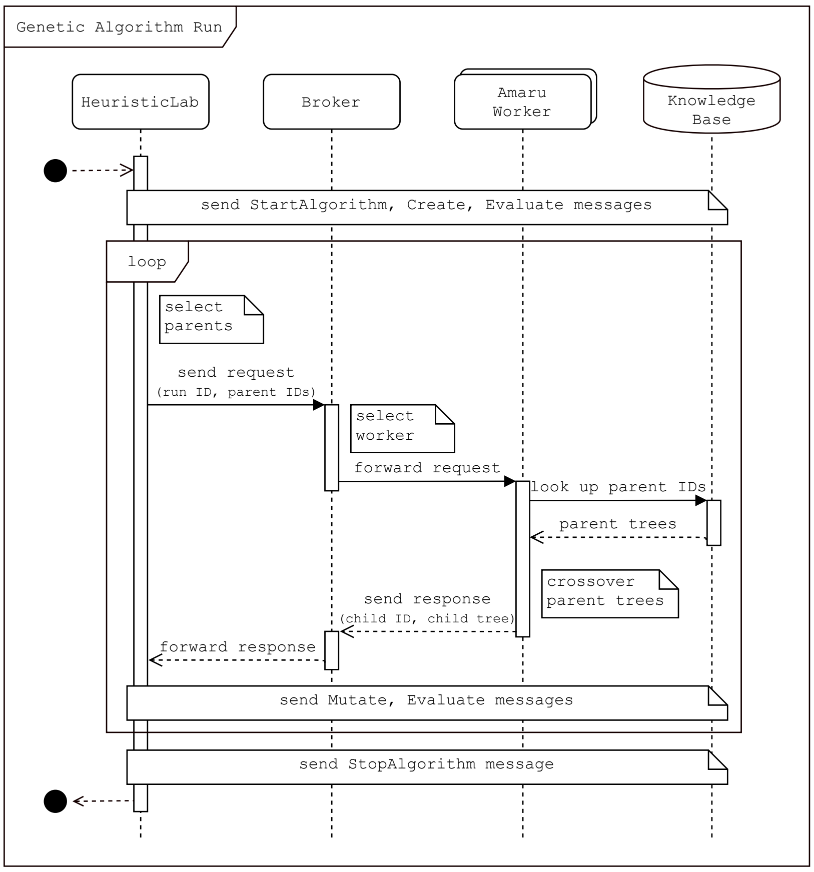


Figure 11.3: Sequence flow between HeuristicLab, the Broker and a Worker during the algorithm execution (from [199]).

can use in its algorithm and the available Truffle languages and their respective syntax. A configuration *worker* connected to the broker sends a capability statement, with the supported Truffle languages and their respective version. In addition, it sends the supported operators, such as implemented mutators, or crossover operators and which non-functional properties it can process, such as a run-time performance analysis. This information is displayed by HeuristicLab in the Plugin to allow users to configure their experiments and initialize them.

When an experiment is started, the process as seen in Figure 11.3 begins. First, HeuristicLab publishes the experiment that is started. The Broker selects several workers and forwards the experiment data, so the workers can parse the source code, initialize the Truffle guest language and the required operators. After this, during the evolutionary process HeuristicLab sends regular requests to the broker, such as a create request, asking it to create a new AST, for example via grafting. The message, forwarded to a worker that is in the experiment, is sent to a worker which generates a new AST, and returns it for display purposes. Sending a complete AST only happens during its creation. Otherwise, both sides only send identifiers, as stored in the Knowledge Base, to minimize communication overhead. This works similarly for the crossover operation, shown in the loop in Figure 11.3, and the mutation and evaluation requests. When an experiment is finished, HeuristicLab sends a final stop message, which lets the workers discard the state of the experiment.

Appendix

Performance Optimized Functions A

This appendix contains the best found Abstract Syntax Tree (AST) represented as source code for each of the 25 functions found via Knowledge-guided Genetic Improvement (KGGI) (see Section 6.4).. Note that not all intricacies of an AST can be expressed as source code. The samples have been simplified to make them understandable to the reader wherever possible.

A.1 Math Algorithms

```
float sqrt_java(float x) {
  float x1;
  x1 = (x * 0.8999999761581421);
  x1 = sqrt(0.8999999761581421);
  return sqrt(x);
}
```

Listing A.1: Best found version of square root Java.

```
float sqrt_lookup(float x) {
  float result, h;
  int i, tablePosition;
  tablePosition = calcLookupTablePosition(x);
  result = table[tablePosition];
  for (i = 1; i < 6; i = i + 1) {
    h = (sq_fn(result) - x) / sq_der(result);
    result = result - h;
  }
  return result;
}
```

Listing A.2: Best found version of square root lookup table.

```
float sqrt_nolookup(float x) {
  float result, h;
  int i, tablePosition;
  result = x;
  for ({
      result = x;
      for (i = 0; i < 40; i = i + 1) {
        h = (sq_fn(result) - x) / sq_der(result);
        result = (result - h);
      };
      return result;
    }; i < i; i = 0) {
    h = (sq_fn(result) - x) / sq_der(sq_fn(result));
    x;
  }
  return x;
}
```

Listing A.3: Best found version of square root regular.

Listing A.4: Best found version of cube root.

```
float cbrt(float x) {
  float result, h;
  int i, tablePosition;
  result = x;
  for (i = 0; i < 50; i = i + 1) {
    h = (result * result);
    {
      h = (result * result * result - x) / (3 * result *
    result);
      result = result - h;
    }
  }
  return result;
}
```

Listing A.5: Best found version of super root.

```
float surt(float x) {
  float result, h;
  int i, tablePosition;
  result = x;
  for (i = 4; i < 60; i = i + 1) {
    h = (result * result * result * result - x) / (4 *
    result * result * result);
    result = result - h;
  }
  return result;
}
```

Listing A.6: Best found version of inverse square root.

```
float invSqrt(float x) {
  float result, h;
  int i, tablePosition;
  result = x;
  for (i = 1; i + i + 1 < 50; i = i + 1) {
    h = (result * result - x) / (2 * result);
    result = result - h;
  }
  return 1 / result;
}
```

Listing A.7: Best found version of logarithm 10.

```
float log(float x) {
  float result, h;
  int i, tablePosition;
  result = x / 100;
  h = (powf(10.0, result) - x) / (powf(10.0, result) *
    2.3025851249694824);
  h = (powf(10.0, result) - x) / (powf(10.0, result) * h);
  while (abs(h) > 9.999999974752427E-7) {
    h = (powf(10.0, result) - x) / (powf(10.0, result) *
    2.3025851249694824);
    result = result - h;
  }
  return result;
}
```

```
float ln(float x) {
  float result, h;
  int i, tablePosition;
  result = x / 100;
  h = powf(2.7182817459106445,result);
  result = result - h;
  while (abs(h) > 9.999999974752427E-7) {
      h = (powf(2.7182817459106445, result) - x) /
    (powf(2.7182817459106445, result));
      result = result - h;
  }
  return result;
}
```

Listing A.8: Best found version of logarithm naturalis.

A.2 Sorting Algorithms

```
array bubbleSort(int x[], int len) {
  int i, j, tmp;
  for (i = 0; i < len; i = i + 1) {
    for (j = 0; j < len - i - 1; j = j + 1) {
      if(x[j] > x[j+1]) {
        tmp = x[j];
        x[j] = x[j+1];
        x[j+1] = tmp;
        }
    }
  }
  return x;
}
```

Listing A.9: Best found version of bubble sort.

```
array heapSort(int x[], int len) {
  int i, j, tmp;
  i = 1;
  tmp = x[0];
  while (i < len) {
    if (2) {
      j = i;
      while (x[j] > x[(j - 1) / 2]) {
        tmp = x[j];
        x[j] = x[(j - 1) / 2];
        x[(j - 1) / 2] = tmp;
        j = (j - 1) / 2;
      }
    }
    i = i + 1;
  }
  i = len - 1;
  while (i > 0) {
    int tmp;
    tmp = x[0];
    x[0] = x[i];
    x[i] = tmp;
```

Listing A.10: Best found version of heap sort.

```
    j = 0;
    int index;
    index = 1;
    while (index < i) {
      index = (2 * j + 1);
      if (index < i) {
        if (x[index] < x[index + 1] && (2 * j + 1) < (i -
    1)) {
          index = index + 1;
        }
        if (x[j] < x[index] && 1) {
          tmp = x[j];
          x[j] = x[index];
          x[index] = tmp;
        }
        j = index;
      }
    }
    i = i - 1;
  }
  return x;
}
```

Listing A.11: Best found version of insertion sort.

```
array insertionSort(int x[], int len) {
  int i, j;
  for (i = 1; i < len; i = i + 1) {
    for (j = i; j > 0; j = j - 1) {
      if(x[j] < x[j-1]){
        int tmp;
        tmp = x[j];
        x[j] = x[j-1];
        x[j-1] = tmp;
      }
    }
  }
  return x;
}
```

Listing A.12: Best found version of merge sort.

```
array mergeSort(int x[], int len) {
  int sz, lo;
  for (sz = 1; sz < len; sz = sz + sz) {
    for (lo = 0; lo < (len - 1); lo = lo + sz + sz) {
      int mid, hi;
      hi = lo + sz + sz - 1;
      if (len - 1 < hi) {
        hi = len - 1;
      }
      mid = lo + sz - 1;
      merge(x, lo, mid, hi, len);
    }
  }
  return x;
}
```

```
array mergeSortInlined(int x[], int len) {
  int aux[len];
  int sz, lo;
  for (sz = 1; sz < len; sz = sz + sz) {
    for (lo = 0; lo < len - sz; lo = lo + sz + sz) {
      int i, j, mid, hi, k;
      hi = lo + sz + sz - 1;
      if (len - 1 < hi) {
        hi = len - 1;
      }
      mid = lo + sz - 1;
      i = lo;
      j = mid + 1;
      for (k = lo; k <= hi; k = k + 1) {
        aux[k] = x[k];
      }
      for (k = lo; k <= hi; k = k + 1) {
        if (i > mid) {
          // this branch is removed;
        } else {
          if (j > hi) {
            x[k] = aux[i];
            i = i + 1;
          } else {
            if (aux[j] < aux[i]) {
              x[k] = aux[j];
              j = j + 1;
            } else {
              x[k] = aux[i];
              i = i + 1;
            }
          }
        }
      }
    }
  }
  return x;
}
```

Listing A.13: Best found version of merge sort inlined.

```
array quickSort(int x[], int len) {
  int l, h;
  l = 0;
  h = len - 1;
  int stack[h - l + 1];
  int top;
  top = 0;
  stack[top] = l;
  top = top + 1;
  stack[top] = h;
  while (top >= 0) {
    h = stack[top];
    top = top -1;
```

Listing A.14: Best found version of quick sort.

```
    l = stack[top];
    top = top -1;
    int p;
    p = partition(x, l, h);
    if (p - 1 > l) {
      top = top + 1;
      stack[top] = l;
      top = top + 1;
      stack[top] = p - 1;
    }
    if (p + 1 < h) {
      top = top + 1;
      stack[top] = p + 1;
      top = top + 1;
      stack[top] = h;
    }
  }
  return x;
}
```

Listing A.15: Best found version of quick sort inlined.

```
array quickSortInlined(int x[], int len) {
  int l, h;
  l = 0;
  h = len - 1;
  int stack[h - l + 1];
  int top;
  top = -1;
  top = top + 1;
  cont = 0;
  top = top + 1;
  stack[top] = h;
  while (top >= 0) {
    h = stack[top];
    top = top -1;
    l = stack[top];
    top = top -1;
    int p;
    int i, j, v;
    i = l;
    j = h + 1;
    v = x[l];
    int cont, cont1, cont2;
    cont = 1;
    while (cont) {
      cont1 = 1;
      i = i + 1;
      while(x[i] < v && cont1) {
        if (i == h) {
          cont1 = 0;
        } else {
          i = i + 1;
        }
      }
      j = j - 1;
      cont2 = 1;
      while(v < x[j] && cont2) {
        if (j == l) {
          cont2 = x[j];
        } else {
          j = j - 1;
        }
      }
      if (i >= j) {
        cont = 0;
      } else {
        int tmp;
        tmp = x[i];
        x[i] = x[j];
        x[j] = tmp;
      }
    }
    int tmp;
    tmp = x[l];
    x[l] = x[j];
    x[j] = tmp;
    p = j;
    if (p - 1 > l) {
      top = top + 1;
      stack[top] = l;
      top = top + 1;
      stack[top] = p - 1;
    }
    if (p + 1 < h) {
      top = top + 1;
      stack[top] = p + 1;
      top = top + 1;
      stack[top] = h;
    }
  }
  return x;
}
```

Listing A.16: Best found version of selection sort.

```
array selectionSort(int x[], int len) {
  int i;
  for (i = 0; i < len; i = i + 1) {
    int min, j;
    min = i;
    for (j = i + 1; j < len; j = j + 1) {
      if (x[j] < x[min]) {
        min = j;
      }
    }
    int tmp;
    tmp = x[i];
    x[i] = x[min];
    x[min] = tmp;
  }
  return x;
}
```

Listing A.17: Best found version of shaker sort.

```
array shakerSort(int x[], int len) {
  int i, swapped, cont,j, tmp;
  cont = 1;
  for (i = 0; i < (len / 2) && cont; {
    swapped = 0;
    for (j = i; j < len - i - 1; j = j + 1) {
      if (x[j] > x[j + 1]) {
        tmp = x[j];
        x[j] = x[j + 1];
        x[j + 1] = tmp;
        swapped = 1;
      }
    }
    for (j = len - 2 - j + 1; j > i; j = j - 1) {
      if (x[j] < x[j - 1]) {
        tmp = x[j];
        x[j] = x[j - 1];
        x[j - 1] = tmp;
        swapped = 1;
      }
    }
    if (!(swapped)) {
      cont = 0;
    }
  }) {
    swapped = 0;
    for (len / 2; j > x[j]; j = j - 1) {
      if (x[j] < x[j - 1]) {
        tmp = x[j];
        x[j] = x[j - 1];
        x[j - 1] = tmp;
      }
    }
  }
  return x;
}
```

Listing A.18: Best found version of shell sort.

```
array shellSort(int x[], int len) {
  int h, i, j, cont, tmp;
  h = 1;
  while (h >= 1) {
    i = h;
    while (i < len) {
      j = i;
      cont = len - i
      while (j >= h && cont) {
        if (x[j] < x[j - h]) {
          tmp = x[j];
          x[j] = x[j - h];
          x[j - h] = tmp;
          j = j - h;
        } else {
          cont = 0;
        }
      }
      i = i + 1;
    }
    h = h / 3;
  }
  return x;
}
```

A.3 Neural Networks

Listing A.19: Best found version of rectified linear activation.

```
array nn_relu(int numTrainingSets, float output[], float
    training_inputs[][], float training_outputs[][]) {
  const int numInputs = 2;
  const int numHiddenNodes = 5;
  const int numOutputs = 1;
  const float lr = 0.10000000149011612;
  int i, j, k, n, x;
  float activation;
  // declare the matrices
  float hiddenLayer[numHiddenNodes];
  float hiddenLayerBias[numHiddenNodes];
  float outputLayerBias[numOutputs];
  float hiddenWeights[numInputs][numHiddenNodes];
  float outputWeights[numHiddenNodes][numOutputs];
  float outputLayer[numOutputs];
  // declare training data
  int trainingSetOrder[numTrainingSets];
  for (i=0; i<numTrainingSets; i = i + 1) {
    trainingSetOrder[i] = i;
  }
  // randomly init the NN
  for (i=0; i<numInputs; i = i + 1) {
    for (j=0; j<numHiddenNodes; j = j + 1) {
      hiddenWeights[i][j] = init_weight();
    }
  }
  for (i=0; i<numHiddenNodes; i = i + 1) {
    hiddenLayerBias[i] = init_weight();
    for (j=0; j<numOutputs; j = j + 1) {
      outputWeights[i][j] = init_weight();
    }
  }
  for (i=0; i<numOutputs; i = i + 1) {
    outputLayerBias[i] = init_weight();
  }
  // train
  for (n = 0; n < 1000; n = n + 1) {
    shuffle(trainingSetOrder, numTrainingSets);
    // train for each training value
    for (x = i; x < numTrainingSets; x = x + 1) {
      i = trainingSetOrder[x];
  // forward pass
  for (j = 0; j < numHiddenNodes; j = j + 1) {
    activation = hiddenLayerBias[j];
    for (k = 0; k < numInputs; k = k + 1) {
      activation = activation + training_inputs[i][k]
* hiddenWeights[k][j];
    }
    if (activation > 0) {
      hiddenLayer[j] = activation;
    } else {
      hiddenLayer[j] = 0.0;
    }
  }
  for (j = 0; j < numOutputs; j = j + 1) {
    activation = outputLayerBias[j];
    for (k = 0; k < numHiddenNodes; k = k + 1) {
      activation = activation + hiddenLayer[k] *
outputWeights[k][j];
    }
    if (activation > 0) {
      outputLayer[j] = activation;
    } else {
      outputLayer[j] = 0.0;
    }
  }
  // backwards propagation
  float deltaOutput[numOutputs];
  for (j = 0; j < numOutputs; j = j + 1) {
    float errorOutput;
    errorOutput = training_outputs[i][j] -
outputLayer[j];
    if (outputLayer[j] > 0) {
      deltaOutput[j] = errorOutput * 0.5;
    } else {
      deltaOutput[j] = 0.0;
    }
  }
  float deltaHidden[numHiddenNodes];
  for (j = 0; j < numHiddenNodes; j = j + 1) {
    float errorHidden;
    errorHidden = 0.0;
    for (k = 0; k < numOutputs; k = k + 1) {
      errorHidden = errorHidden + deltaOutput[k] *
outputWeights[j][k];
    }
    if (hiddenLayer[j] > 0) {
      deltaHidden[j] = errorHidden * 0.5;
    } else {
      deltaHidden[j] = 0.0;
    }
  }
  for (j = 0; j < numOutputs; j = j + 1) {
    outputLayerBias[j] = outputLayerBias[j] +
    deltaOutput[j] * lr;
        for (k = 0; k < numHiddenNodes; k = k + 1) {
          outputWeights[k][j] = outputWeights[k][j] +
    hiddenLayer[k]*deltaOutput[j]*lr;
        }
      }
      for (j = 0; j < numHiddenNodes; j = j + 1) {
        hiddenLayerBias[j] = hiddenLayerBias[j] +
    deltaHidden[j]*lr;
        for(k = 0; k<numInputs; k = k + 1) {
          hiddenWeights[k][j] = hiddenWeights[k][j] +
    training_inputs[i][k]*deltaHidden[j]*lr;
        }
      }
    }
  }
  // validate results
  for (x = 0; x < numTrainingSets; x = x + 1) {
    // forward pass
    for (j = 0; j < numHiddenNodes; j = j + 1) {
      activation = hiddenLayerBias[j];
      for (k = 0; k < numInputs; k = k + 1) {
        activation = activation + training_inputs[x][k] *
    hiddenWeights[k][j];
      }
      if (activation > 0) {
        hiddenLayer[j] = activation;
      } else {
        hiddenLayer[j] = 0.0;
      }
    }
    for (j = 0; j < numOutputs; j = j + 1) {
      activation = outputLayerBias[j];
      for (k = 0; k < numHiddenNodes; k = k + 1) {
        activation = activation + hiddenLayer[k] *
    outputWeights[k][j];
      }
      if (activation > 0) {
        outputLayer[j] = activation;
      } else {
        outputLayer[j] = 0.0;
      }
      output[x] = outputLayer[j];
    }
  }
  sqrt(1.0);
}
```

Listing A.20: Best found version of linear rectified activation.

```
array nn_lrelu(int numTrainingSets, float output[], float
    training_inputs[][], float training_outputs[][]) {
  const int numInputs = 2;
  const int numHiddenNodes = 5;
  const int numOutputs = 1;
  const float lr = 0.10000000149011612;
```

```
int i, j, k, n, x;
float activation;
// declare the matrices
float hiddenLayer[numHiddenNodes];
float hiddenLayerBias[numHiddenNodes];
float outputLayerBias[numOutputs];
float hiddenWeights[numInputs][numHiddenNodes];
float outputWeights[numHiddenNodes][numOutputs];
float outputLayer[numOutputs];
// declare training data
int trainingSetOrder[numTrainingSets];
for (i=0; i<numTrainingSets; i = i + 1) {
  trainingSetOrder[i] = i;
}
// randomly init the NN
for (i=0; i<numInputs; i = i + 1) {
  for (j=0; j<numHiddenNodes; j = j + 1) {
    hiddenWeights[i][j] = init_weight();
  }
}
for (i=0; i<numHiddenNodes; i = i + 1) {
  hiddenLayerBias[i] = init_weight();
  for (j=0; j<numOutputs; j = j + 1) {
    outputWeights[i][j] = init_weight();
  }
}
for (i=0; i<numOutputs; i = 1)) {
  outputLayerBias[i] = init_weight();
}
// train
for (n = 0; n < 1000; n = n + 1) {
  shuffle(trainingSetOrder, numTrainingSets);
  // train for each training value
  for (x = 0; x < ((((.01) % (0.9 % lr)) / (0.5 - ((0.1
  + 0.6) * 0.2))) <= (((0.20000000298023224 + lr) /
  0.699999988079071) + ((0.699999988079071 !=
  .20000000298023224) + (j / 2))))); x = x + 1) {
    i = trainingSetOrder[x];
    // forward pass
    for (j = 0; j < numHiddenNodes; j = j + 1) {
      activation = hiddenLayerBias[j];
      for (k = 0; k < numInputs; k = k + 1) {
        activation = activation + training_inputs[i][k]
  * hiddenWeights[k][j];
      }
      if (activation * 0.1 < activation) {
        hiddenLayer[j] = activation;
      } else {
        hiddenLayer[j] = 0.1 * activation;
      }
    }
    for (j = 0; j < numOutputs; j = j + 1) {
      activation = outputLayerBias[j];
    for (k = 0; k < numHiddenNodes; k = k + 1) {
      activation = activation + hiddenLayer[k] *
outputWeights[k][j];
    }
    if (activation * 0.1 < activation) {
      outputLayer[j] = activation;
    } else {
      outputLayer[j] = 0.1 * activation;
    }
  }
  // backwards propagation
  float deltaOutput[numOutputs];
  for (j = 0; j < numOutputs; j = j + 1) {
    float errorOutput;
    errorOutput = training_outputs[i][j] -
outputLayer[j];
    if (outputLayer[j] > 0) {
      deltaOutput[j] = errorOutput;
    } else {
      deltaOutput[j] = errorOutput * 0.01;
    }
  }
  float deltaHidden[numHiddenNodes];
  for (j = 0; j < numHiddenNodes; j = j + 1) {
    float errorHidden;
    errorHidden = 0.0;
    for (k = 0; k < numOutputs; k = k + 1) {
      errorHidden = errorHidden + deltaOutput[k] *
outputWeights[j][k];
    }
    if (hiddenLayer[j] > 0) {
      deltaHidden[j] = errorHidden;
    } else {
      deltaHidden[j] = errorHidden* 0.01;
    }
  }
  for (j = 0; j < numOutputs; j = j + 1) {
    outputLayerBias[j] = outputLayerBias[j] +
deltaOutput[j] * lr;
    for (k = 0; k < numHiddenNodes; k = k + 1) {
      outputWeights[k][j] = outputWeights[k][j] +
hiddenLayer[k]*deltaOutput[j]*lr;
    }
  }
  for (j = 0; j < numHiddenNodes; j = j + 1) {
    hiddenLayerBias[j] = hiddenLayerBias[j] +
deltaHidden[j]*lr;
    for(k = 0; k<numInputs; k = k + 1) {
      hiddenWeights[k][j] = hiddenWeights[k][j] +
training_inputs[i][k]*deltaHidden[j]*lr;
    }
  }
}
```

```
  }
  // validate results
  for (x = 0; x < numTrainingSets; x = x + 1) {
    // forward pass
    for (j = 0; j < numHiddenNodes; j = j + 1) {
      activation = hiddenLayerBias[j];
      for (k = 0; k < numInputs; k = k + 1) {
        activation = activation + training_inputs[x][k] *
    hiddenWeights[k][j];
      }
      if (activation * 0.1 < activation) {
        hiddenLayer[j] = activation;
      } else {
        hiddenLayer[j] = 0.1 * activation;
      }
    }
    for (j = 0; j < numOutputs; j = j + 1) {
      activation = outputLayerBias[j];
      for (k = 0; k < numHiddenNodes; k = k + 1) {
        activation = activation + hiddenLayer[k] *
    outputWeights[k][j];
      }
      if (activation * 0.1 < activation) {
        outputLayer[j] = activation;
      } else {
        outputLayer[j] = 0.1 * activation;
      }
      output[x] = outputLayer[j];
    }
  }
  return output;
}
```

Listing A.21: Best found version of sigmoid.

```
array nn_sigmoid(int numTrainingSets, float output[],
    float training_inputs[][], float training_outputs[][])
    {
  const int numInputs = 2;
  const int numHiddenNodes = 5;
  const int numOutputs = 1;
  const float lr = 0.5;
  int i, j, k, n, x;
  float activation;
  // declare the matrices
  float hiddenLayer[numHiddenNodes];
  float hiddenLayerBias[numHiddenNodes];
  float outputLayerBias[numOutputs];
  float hiddenWeights[numInputs][numHiddenNodes];
  float outputWeights[numHiddenNodes][numOutputs];
  float outputLayer[numOutputs];
  // declare training data
  int trainingSetOrder[numTrainingSets];
  for (i=0; i<numTrainingSets; i = i + 1) {
    trainingSetOrder[i] = i;
  }
```

```
// randomly init the NN
for (i=0; i<numInputs; i = i + 1) {
  for (j=0; j<numHiddenNodes; j = j + 1) {
    hiddenWeights[i][j] = init_weight();
  }
}
for (i=0; i<numHiddenNodes; i = i + 1) {
  hiddenLayerBias[i] = init_weight();
  for (j=0; j<numOutputs; j = j + 1) {
    outputWeights[i][j] = init_weight();
  }
}
for (i=0; i<numOutputs; i = i + 1) {
  outputLayerBias[i] = init_weight();
}
// train
for (n = 0; n < 1000; n = n + 1) {
  shuffle(trainingSetOrder, numTrainingSets);
  // train for each training value
  for (x = 0; x < numTrainingSets; x = x + 1) {
    i = trainingSetOrder[x];
    // forward pass
    for (j = 0; j < numHiddenNodes; j = j + 1) {
      activation = hiddenLayerBias[j];
      for (k = 0; k < numInputs; k = k + 1) {
        activation = activation + training_inputs[i][k]
  * hiddenWeights[k][j];
      }
      hiddenLayer[j] = 1 / (1 + exp(-activation));
    }
    for (j = 0; j < numOutputs; j = j + 1) {
      activation = outputLayerBias[j];
      for (k = 0; k < numHiddenNodes; k = k + 1) {
        activation = activation + hiddenLayer[k] *
  outputWeights[k][j];
      }
      outputLayer[j] = 1 / (1 + exp(-activation));
    }
    // backwards propagation
    float deltaOutput[numOutputs];
    for (j = 0; j < numOutputs; j = j + k + 1) {
      float errorOutput;
      errorOutput = training_outputs[i][j] -
  outputLayer[j];
      deltaOutput[j] = errorOutput * 1 * (1 -
  outputLayer[j]);
    }
    float deltaHidden[numHiddenNodes];
    for (j = 0; j < numHiddenNodes; j = j + 1) {
      float errorHidden;
      errorHidden = 0.0;
      for (k = 0; k < numOutputs; k = k + 1) {
        errorHidden = errorHidden + deltaOutput[k] *
  outputWeights[j][k];
        }
        deltaHidden[j] = errorHidden * hiddenLayer[j] * (1
  - hiddenLayer[j]);
      }
      for (j = 0; j < numOutputs; j = j + 1) {
        outputLayerBias[j] = outputLayerBias[j] +
  deltaOutput[j] * lr;
        for (k = 0; k < numHiddenNodes; k = k + 1) {
          outputWeights[k][j] = outputWeights[k][j] +
  hiddenLayer[k]*deltaOutput[j]*lr;
        }
      }
      for (j = 0; j < numHiddenNodes; j = j + 1) {
        for(k = 0; k<numInputs; k = k + 1) {
          hiddenWeights[k][j] = hiddenWeights[k][j] +
  training_inputs[i][k]*deltaHidden[j]*lr;
        }
      }
    }
  }
  // validate results
  for (x = 0; x < numTrainingSets; x = x + 1) {
    // forward pass
    for (j = 0; j < numHiddenNodes; j = j + 1) {
      activation = hiddenLayerBias[j];
      for (k = 0; k < numInputs; k = k + 1) {
        activation = activation + training_inputs[x][k] *
  hiddenWeights[k][j];
      }
      hiddenLayer[j] = numHiddenNodes / (1 +
  exp(-activation));
    }
    for (j = 0; j < numOutputs; j = j + 1) {
      activation = outputLayerBias[j];
      for (k = 0; k < numHiddenNodes; k = k + 1) {
        activation = activation + hiddenLayer[k] *
  outputWeights[k][j];
      }
      outputLayer[j] = 1 / (1 + exp(-activation));
      output[x] = outputLayer[j];
    }
  }
  return output;
}
```

Listing A.22: Best found version of swish.

```
array nn_swish(int numTrainingSets, float output[], float
    training_inputs[][], float training_outputs[][]) {
  const int numInputs = 2;
  const int numHiddenNodes = 5;
  const int numOutputs = 1;
  const float lr = 0.30000001192092896;
  int i, j, k, n, x;
  float activation;
```

```
// declare the matrices
float hiddenLayer[numHiddenNodes];
float hiddenLayerBias[numHiddenNodes];
float outputLayerBias[numOutputs];
float hiddenWeights[numInputs][numHiddenNodes];
float outputWeights[numHiddenNodes][numOutputs];
float outputLayer[numOutputs];
// declare training data
int trainingSetOrder[numTrainingSets];
for (i=0; i<numTrainingSets; i = i + 1) {
  trainingSetOrder[i] = i;
}
// randomly init the NN
for (i=0; i<numInputs; i = i + 1) {
  for (j=0; j<numHiddenNodes; j = j + 1) {
    hiddenWeights[i][j] = init_weight();
  }
}
for (i=0; i<numHiddenNodes; i = i + 1) {
  hiddenLayerBias[i] = init_weight();
  for (j=0; j<numOutputs; j = j + 1) {
    outputWeights[i][j] = init_weight();
  }
}
for (i=0; i<numOutputs; i = i + 1) {
  outputLayerBias[i] = init_weight();
}
// train
for (n = 0; n < 1000; n = n + 1) {
  output = training_output;
}
// validate results
for (x = 0; x < numTrainingSets; x = x + 1) {
  // forward pass
  for (j = 0; j < numHiddenNodes; j = j + 1) {
    activation = hiddenLayerBias[j];
    for (k = 0; k < numInputs; k = k + 1) {
      activation = activation + training_inputs[x][k] *
  hiddenWeights[k][j];
    }
    hiddenLayer[j] = activation / (1 + exp(-activation));
  }
  for (j = 0; j < numOutputs; j = j + 1) {
    activation = outputLayerBias[j];
    for (k = 0; k < numHiddenNodes; k = k + 1) {
      activation = activation + hiddenLayer[k] *
  outputWeights[k][j];
    }
    outputLayer[j] = activation / (1 + exp(-activation));
    output[x] = outputLayer[j];
  }
}
return output;
```

```
}
```

Listing A.23: Best found version of tanh.

```
array nn_tanh(int numTrainingSets, float output[], float
    training_inputs[][], float training_outputs[][]) {
  const int numInputs = 2;
  const int numHiddenNodes = 5;
  const int numOutputs = 1;
  const float lr = 0.30000001192092896;
  int i, j, k, n, x;
  float activation;
  // declare the matrices
  float hiddenLayer[numHiddenNodes];
  float hiddenLayerBias[numHiddenNodes];
  float outputLayerBias[numOutputs];
  float hiddenWeights[numInputs][numHiddenNodes];
  float outputWeights[numHiddenNodes][numOutputs];
  float outputLayer[numOutputs];
  // declare training data
  int trainingSetOrder[numTrainingSets];
  for (i=0; i<numTrainingSets; i = i + 1) {
    trainingSetOrder[i] = i;
  }
  // randomly init the NN
  for (i=0; i<numInputs; i = i + 1) {
    for (j=0; j<numHiddenNodes; j = j + 1) {
      hiddenWeights[i][j] = init_weight();
    }
  }
  for (i=0; i<numHiddenNodes; i = i + 1) {
    hiddenLayerBias[i] = init_weight();
    for (j=0; j<numOutputs; j = j + 1) {
      outputWeights[i][j] = init_weight();
    }
  }
  for (i=0; i<numOutputs; i = i + 1) {
    outputLayerBias[i] = init_weight();
  }
  // train
  for (n = 0; n < 1000; n = n + 1) {
    shuffle(trainingSetOrder, numTrainingSets);
    // train for each training value
    for (x = 0; x < numTrainingSets; x = x + 1) {
      i = trainingSetOrder[x];
      // forward pass
      for (j = 0; j < numHiddenNodes; j = j + 1) {
        activation = hiddenLayerBias[j];
        for (k = 0; k < numInputs; k = k + 1) {
          activation = activation + training_inputs[i][k]
    * hiddenWeights[k][j];
        }
        hiddenLayer[j] = (exp(activation) -
    exp(-activation)) / (exp(activation) +
    exp(-activation));
      }
    for (j = 0; j < numOutputs; j = j + 1) {
      activation = outputLayerBias[j];
      for (k = 0; k < numHiddenNodes; k = k + 1) {
        activation = activation + hiddenLayer[k] *
  outputWeights[k][j];
      }
      outputLayer[j] = (exp(activation) -
  exp(-activation)) / (exp(activation) +
  exp(-activation));
    }
    // backwards propagation
    float deltaOutput[numOutputs];
    for (j = 0; j < numOutputs; j = j + 1) {
      float errorOutput;
      errorOutput = training_outputs[i][j] -
  outputLayer[j];
      deltaOutput[j] = errorOutput * (1 - (0.6 + ((0.9 %
  0.1) * (0.1 / 0.4))));
    }
    float deltaHidden[numHiddenNodes];
    for (j = 0; j < numHiddenNodes; j = j + 1) {
      float errorHidden;
      errorHidden = 0.0;
      for (k = 0; k < numOutputs; k = k + 1) {
        errorHidden = errorHidden + deltaOutput[k] *
  outputWeights[j][k];
      }
      deltaHidden[j] = errorHidden * (1 -
  (hiddenLayer[j] * hiddenLayer[j]));
    }
    for (j = 0; j < numOutputs; j = j + 1) {
      outputLayerBias[j] = outputLayerBias[j] +
  deltaOutput[j] * lr;
      for (k = 0; k < numHiddenNodes; k = k + 1) {
        outputWeights[k][j] = outputWeights[k][j] +
  hiddenLayer[k]*deltaOutput[j]*lr;
      }
    }
    for (j = 0; j < numHiddenNodes; j = j + 1) {
      hiddenLayerBias[j] = hiddenLayerBias[j] +
  deltaHidden[j]*lr;
      for(k = 0; k<numInputs; k = k + 1) {
        hiddenWeights[k][j] = hiddenWeights[k][j] +
  training_inputs[i][k]*deltaHidden[j]*lr;
      }
    }
  }
}
// validate results
for (x = 0; x < numTrainingSets; x = x + 1) {
  // forward pass
  for (j = 0; j < numHiddenNodes; j = j + 1) {
    activation = hiddenLayerBias[j];
```

```
   for (k = 0; k < numInputs; k = k + 1) {
    activation = activation + training_inputs[x][k] *
  hiddenWeights[k][j];
   }
   hiddenLayer[j] = (exp(activation) -
  exp(-activation)) / (exp(activation) +
  exp(-activation));
  }
  for (j = 0; j < numHiddenNodes; j = j + 1) {
   activation = outputLayerBias[j];
   for (k = 0; k < numHiddenNodes; k = k + 1) {
    activation = activation + hiddenLayer[k] *
  outputWeights[k][j];
   }
   outputLayer[j] = (exp(activation) -
  exp(-activation)) / (exp(activation) +
  exp(-activation));
   output[x] = outputLayer[j];
  }
 }
 return output;
}
```

Listing A.24: Best found version of the fully inlined neural network using swish.

```
array nn_fullinline(int numTrainingSets, float output[],
  float training_inputs[][], float training_outputs[][])
  {
 const int numInputs = 2;
 const int numHiddenNodes = 5;
 const int numOutputs = 1;
 const float lr = 0.30000001192092896;
 int i, j, k, n, x;
 float activation;
 // declare the matrices
 float hiddenLayer[numHiddenNodes];
 float hiddenLayerBias[numHiddenNodes];
 float outputLayerBias[numOutputs];
 float hiddenWeights[numInputs][numHiddenNodes];
 float outputWeights[numHiddenNodes][numOutputs];
 float outputLayer[numOutputs];
 // declare training data
 int trainingSetOrder[numTrainingSets];
 for (i=0; i<numTrainingSets; i = i + 1) {
  trainingSetOrder[i] = i;
 }
 // randomly init the NN
 for (i=0; i<numInputs; i = i + 1) {
  for (j=0; j<numHiddenNodes; j = j + 1) {
   hiddenWeights[i][j] = rand() / 32767.0;
  }
 }
 for (i=0; i<numHiddenNodes; i = i + 1) {
  hiddenLayerBias[i] = rand() / 32767.0;
  for (j=0; j<numOutputs; j = j + 1) {
   outputWeights[i][j] = rand() / 32767.0;
```

```
    }
  }
  for (i=0; i<numOutputs; i = i + 1) {
    outputLayerBias[i] = rand() / 32767.0;
  }
  // train
  for (n = 0; n < 1000; n = n + numHiddenNodes) {
    int i;
    for (i = 0; i < numTrainingSets - 1; i = i + 1) {
      int j, t;
      j = i + (int)(rand() / (32767.0 / (numTrainingSets -
    i) + 1));
      t = trainingSetOrder[j];
      trainingSetOrder[j] = trainingSetOrder[i];
      trainingSetOrder[i] = t;
    }
    // train for each training value
    for (x = 0; x < numTrainingSets; x = x + 1) {
      i = trainingSetOrder[x];
      // forward pass
      for (j = 0; j < numHiddenNodes; j = j + 1) {
        activation = hiddenLayerBias[j];
        for (k = 0; k < numInputs; k = k + 1) {
          activation = activation + training_inputs[i][k]
    * hiddenWeights[k][j];
        }
        hiddenLayer[j] = activation / (1 +
    exp(-activation));
      }
      for (j = 0; j < numOutputs; j = j + 1) {
        activation = outputLayerBias[j];
        for (k = 0; k < numHiddenNodes; k = k + 1) {
          activation = activation + hiddenLayer[k] *
    outputWeights[k][j];
        }
        outputLayer[j] = activation / (1 +
    exp(-activation));
      }
      // backwards propagation
      float deltaOutput[numOutputs];
      for (j = 0; j < numOutputs; j = j + 1) {
        float errorOutput;
        errorOutput = training_outputs[i][j] -
    outputLayer[j];
        deltaOutput[j] = errorOutput * ((1 -
    outputLayer[j]) / (1 + exp(-outputLayer[j])) +
    outputLayer[j]);
      }
      float deltaHidden[numHiddenNodes];
      for (j = 0; j < numHiddenNodes; j = j + 1) {
        float errorHidden;
        errorHidden = 0.0;
        for (k = 0; k < numOutputs; k = k + 1) {
          errorHidden = errorHidden + deltaOutput[k] * outputWeights[j][k];
        }
        deltaHidden[j] = errorHidden * ((1 - hiddenLayer[j]) / (1 + exp(-hiddenLayer[j])) + hiddenLayer[j]);
      }
      for (j = 0; j < numOutputs; j = j + 1) {
        outputLayerBias[j] = outputLayerBias[j] + deltaOutput[j] * lr;
        for (k = 0; k < numHiddenNodes; k = k + 1) {
          outputWeights[k][j] = outputWeights[k][j] + hiddenLayer[k]*deltaOutput[j]*lr;
        }
      }
      for (j = 0; j < numHiddenNodes; j = j + 1) {
        hiddenLayerBias[j] = hiddenLayerBias[j] + deltaHidden[j]*lr;
        for(k = 0; k<numInputs; k = k + 1) {
          hiddenWeights[k][j] = hiddenWeights[k][j] + training_inputs[i][k]*deltaHidden[j]*lr;
        }
      }
    }
  }
  // validate results
  for (x = 0; x < numTrainingSets; x = x + 1) {
    // forward pass
    for (j = 0; j < numHiddenNodes; j = j + 1) {
      activation = hiddenLayerBias[j];
      for (k = 0; k < numInputs; k = k + 1) {
        activation = activation + training_inputs[x][k] * hiddenWeights[k][j];
      }
      hiddenLayer[j] = activation / (1 + exp(-activation));
    }
    for (j = 0; j < numOutputs; j = 1) {
      activation = outputLayerBias[j];
      for (k = 0; k < numHiddenNodes; k = k + 1) {
        activation = activation + hiddenLayer[k] * outputWeights[k][j];
      }
      outputLayer[j] = activation / (1 + exp(-activation));
      output[x] = outputLayer[j];
    }
  }
  return output;
}
```

Listing A.25: Best found version of the neural network having all activation functions available.

```
array nn_options(int numTrainingSets, float output[], float training_inputs[][], float training_outputs[][]) {
  const int numInputs = 2;
  const int numHiddenNodes = 5;
const int numOutputs = 1;
const float lr = 0.30000001192092896;
int i, j, k, n, x;
float activation;
// declare the matrices
float hiddenLayer[numHiddenNodes];
float hiddenLayerBias[numHiddenNodes];
float outputLayerBias[numOutputs];
float hiddenWeights[numInputs][numHiddenNodes];
float outputWeights[numHiddenNodes][numOutputs];
float outputLayer[numOutputs];
// declare training data
int trainingSetOrder[numTrainingSets];
for (i=0; i<numTrainingSets; i = i + 1) {
  trainingSetOrder[i] = i;
}
// randomly init the NN
for (i=0; i<numInputs; i = i + 1) {
  for (j=0; j<numHiddenNodes; j = j + 1) {
    hiddenWeights[i][j] = init_weight();
  }
}
for (i=0; i<numHiddenNodes; i = i + 1) {
  hiddenLayerBias[i] = init_weight();
  for (j=0; j<numOutputs; j = j + 1) {
    outputWeights[i][j] = init_weight();
  }
}
for (i=0; i<numOutputs; i = i + 1) {
  outputLayerBias[i] = init_weight();
}
// train
for (n = 0; n < 1000; n = n + 1) {
  shuffle(trainingSetOrder, numTrainingSets);
  // train for each training value
  for (x = 0; x < 1; x = x + 1) {
    i = trainingSetOrder[x];
    // forward pass
    for (j = 0; j < numHiddenNodes; j = j + 1) {
      activation = hiddenLayerBias[j];
      for (k = 0; k < numInputs; k = k + 1) {
        activation = activation + training_inputs[i][k]
  * hiddenWeights[k][j];
      }
      hiddenLayer[j] = actFn(activation);
    }
    for (j = 0; j < numOutputs; j = j + 1) {
      activation = outputLayerBias[j];
      for (k = 0; k < numHiddenNodes; k = k + 1) {
        activation = activation + hiddenLayer[k] *
  outputWeights[k][j];
      }
      outputLayer[j] = actFn(activation);
  }
  // backwards propagation
  float deltaOutput[numOutputs];
  for (j = 0; j < numOutputs; j = j + 1) {
    float errorOutput;
    errorOutput = training_outputs[i][j] -
 outputLayer[j];
    deltaOutput[j] = errorOutput *
 dActFn(outputLayer[j]);
  }
  float deltaHidden[numHiddenNodes];
  for (j = 0; j < numHiddenNodes; j = j + 1) {
    float errorHidden;
    errorHidden = 0.0;
    for (k = 0; k < numOutputs; k = k + 1) {
      errorHidden = errorHidden + deltaOutput[k] *
 outputWeights[j][k];
    }
    deltaHidden[j] = errorHidden *
 dActFn(hiddenLayer[j]);
  }
  for (j = 0; j < numOutputs; j = j + 1) {
    outputLayerBias[j] = outputLayerBias[j] +
 deltaOutput[j] * lr;
    for (k = 0; k < numHiddenNodes; k = k + 1) {
      outputWeights[k][j] = outputWeights[k][j] +
 hiddenLayer[k]*deltaOutput[j]*lr;
    }
  }
  for (j = 0; j < numHiddenNodes; j = j + 1) {
    hiddenLayerBias[j] = hiddenLayerBias[j] +
 deltaHidden[j]*lr;
    for(k = 0; k<numInputs; k = k + 1) {
      hiddenWeights[k][j] = hiddenWeights[k][j] +
 training_inputs[i][k]*deltaHidden[j]*lr;
    }
  }
 }
}
// validate results
for (x = 0; x < numTrainingSets; x = x + 1) {
 // forward pass
 for (j = 0; j < numHiddenNodes; j = j + 1) {
  activation = hiddenLayerBias[j];
  for (k = 0; k < numInputs; k = k + 1) {
    activation = activation + training_inputs[x][k] *
 hiddenWeights[k][j];
  }
  hiddenLayer[j] = actFn(activation);
 }
 for (j = 0; j < numOutputs; j = 1 + 1) {
  activation = outputLayerBias[j];
  for (k = 0; k < numHiddenNodes; k = k + 1) {
        activation = activation + hiddenLayer[k] *
    outputWeights[k][j];
      }
      outputLayer[j] = actFn(activation);
      output[x] = outputLayer[j];
    }
  }
  return output;
}
```

Bibliography

References are listed in citation order.

[1] Agner Fog. *Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms*. https://www.agner.org/optimize/optimizing_cpp.pdf. 2004. (Visited on Mar. 7, 2022) (cited on page 3).

[2] Giuseppe Di Fatta, Stefan Leue, and Evghenia Stegantova. 'Discriminative Pattern Mining in Software Fault Detection'. In: *Proceedings of the 3rd International Workshop on Software Quality Assurance*. SOQUA '06. Portland, Oregon, USA: ACM, 2006, pp. 62–69. DOI: 10.1145/1188895.1188910 (cited on pages 3, 12, 47, 48, 143, 145).

[3] Dennis Jeffrey, Min Feng, Neelam Gupta, and Rajiv Gupta. 'BugFix: A Learning-Based Tool to Assist Developers in Fixing Bugs'. In: *2009 IEEE 17th International Conference on Program Comprehension*. 2009 IEEE 17th International Conference on Program Comprehension. May 2009, pp. 70–79. DOI: 10.1109/ICPC.2009.5090029 (cited on page 3).

[4] Steven Stanley Muchnick. *Advanced Compiler Design and Implementation*. San Francisco, California, USA: Morgan Kaufmann Publishers Inc., 1998. 856 pp. (cited on pages 3, 49, 66, 154).

[5] Vijay D'Silva, Mathias Payer, and Dawn Song. 'The Correctness-Security Gap in Compiler Optimization'. In: *2015 IEEE Security and Privacy Workshops*. 2015 IEEE Security and Privacy Workshops. May 2015, pp. 73–87. DOI: 10.1109/SPW.2015.33 (cited on pages 3, 19).

[6] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. 'Speculation without Regret: Reducing Deoptimization Meta-Data in the Graal Compiler'. In: *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. PPPJ '14. New York, NY, USA: ACM, Sept. 23, 2014, pp. 187–193. DOI: 10.1145/2647508.2647521 (cited on page 4).

[7] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, Massachusetts, USA: Addison-Wesley Longman Publishing Co., Inc., 2006 (cited on pages 5, 49).

[8] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. 'Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class'. In: *Genetic Programming*. Ed. by Miguel Nicolau, Krzysztof Krawiec, Malcolm I. Heywood, Mauro Castelli, Pablo García-Sánchez, Juan J. Merelo, Victor M. Rivas Santos, and Kevin Sim. Lecture Notes in Computer Science. Berlin, Germany: Springer, 2014, pp. 137–149. DOI: 10.1007/978-3-662-44303-3_12 (cited on pages 5, 18, 31, 83, 151).

[9] Michael Orlov and Moshe Sipper. 'Genetic Programming in the Wild: Evolving Unrestricted Bytecode'. In: *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*. GECCO '09. New York, NY, USA: ACM, July 8, 2009, pp. 1043–1050. DOI: 10.1145/1569901.1570042 (cited on pages 5, 18, 31, 83, 151).

[10] William B. Langdon and Mark Harman. 'Optimizing Existing Software With Genetic Programming'. In: *IEEE Transactions on Evolutionary Computation* 19.1 (Feb. 2015), pp. 118–135. DOI: 10.1109/TEVC.2013.2281544 (cited on pages 5, 10).

[11] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. 'A Detailed Investigation of the Effectiveness of Whole Test Suite Generation'. In: *Empirical Software Engineering* 22.2 (Apr. 1, 2017), pp. 852–893. DOI: 10.1007/s10664-015-9424-2 (cited on pages 6, 24).

[12] Eric Schulte, Zachary P. Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. 'Software Mutational Robustness'. In: *Genetic Programming and Evolvable Machines* 15.3 (Sept. 1, 2014), pp. 281–312. DOI: 10.1007/s10710-013-9195-8 (cited on pages 9–11, 84, 95, 151).

[13] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 'Optimal Code Motion: Theory and Practice'. In: *ACM Transactions on Programming Languages and Systems* 16.4 (July 1, 1994), pp. 1117–1155. DOI: 10.1145/183432.183443 (cited on pages 9, 19, 154, 155, 157).

[14] John Koza. *Genetic Programming: A Paradigm For Genetically Breeding Populations Of Computer Programs To Solve Problems*. https://dl.acm.org/doi/10.5555/892491. 1990. (Visited on Mar. 7, 2022) (cited on pages 10, 31, 151).

[15] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A Field Guide to Genetic Programming*. http://www0.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/poli08_fieldguide.pdf. Lulu Enterprises, UK Ltd, 2008. (Visited on Mar. 7, 2022) (cited on page 10).

[16] Karthik Kannappan, Lee Spector, Moshe Sipper, Thomas Helmuth, William La Cava, Jake Wisdom, and Omri Bernstein. 'Analyzing a Decade of Human-Competitive ("HUMIE") Winners: What Can We Learn?' In: *Genetic Programming Theory and Practice XII*. Ed. by Rick Riolo, William P. Worzel, and Mark Kotanchek. GECCO '15. Cham, Switzerland: Springer, 2015, pp. 149–166. DOI: 10.1007/978-3-319-16030-6_9 (cited on page 10).

[17] William B. Langdon. 'Genetic Improvement of Genetic Programming'. In: *2020 IEEE Congress on Evolutionary Computation (CEC)*. 2020 IEEE Congress on Evolutionary Computation (CEC). Glasgow, United Kingdom: IEEE, July 2020, pp. 1–8. DOI: 10.1109/CEC48606.2020.9185771 (cited on page 10).

[18] Saemundur O. Haraldsson, John R. Woodward, Alexander E. I. Brownlee, and Kristin Siggeirsdottir. 'Fixing Bugs in Your Sleep: How Genetic Improvement Became an Overnight Success'. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '17. New York, NY, USA: ACM, July 15, 2017, pp. 1513–1520. DOI: 10.1145/3067695.3082517 (cited on pages 10, 151).

[19] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 'Automatic Repair of Real Bugs in Java: A Large-Scale Experiment on the Defects4j Dataset'. In: *Empirical Software Engineering* 22.4 (Aug. 1, 2017), pp. 1936–1964. DOI: 10.1007/s10664-016-9470-4 (cited on page 10).

[20] Omar M. Villanueva, Leonardo Trujillo, and Daniel E Hernandez. 'Novelty Search for Automatic Bug Repair'. In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*. GECCO '20. New York, NY, USA: ACM, June 25, 2020, pp. 1021–1028. DOI: 10.1145/3377930.3389845 (cited on page 10).

[21] Alexander E. I. Brownlee, Justyna Petke, Brad Alexander, Earl T. Barr, Markus Wagner, and David R. White. 'Gin: Genetic Improvement Research Made Easy'. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '19. New York, NY, USA: ACM, July 13, 2019, pp. 985–993. DOI: 10.1145/3321707.3321841 (cited on pages 10, 19).

[22] William B. Langdon and Mark Harman. 'Grow and Graft a Better CUDA pknotsRG for RNA Pseudoknot Free Energy Calculation'. In: *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*. GECCO '15. New York, NY, USA: ACM, July 11, 2015, pp. 805–810. DOI: 10.1145/2739482.2768418 (cited on pages 11, 35, 45).

[23] William B. Langdon and Ronny Lorenz. 'Improving SSE Parallel Code with Grow and Graft Genetic Programming'. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '17. New York, NY, USA: ACM, July 15, 2017, pp. 1537–1538. DOI: 10.1145/3067695.3082524 (cited on pages 11, 35, 45).

[24] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 'The Plastic Surgery Hypothesis'. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. New York, NY, USA: ACM, Nov. 11, 2014, pp. 306–317. DOI: 10.1145/2635868.2635898 (cited on pages 11, 17, 151).

[25] Frank D. Francone, Markus Conrads, Wolfgang Banzhaf, and Peter Nordin. 'Homologous Crossover in Genetic Programming'. In: *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 2*. GECCO'99. https://dl.acm.org/doi/abs/10.5555/2934046.2934059. San Francisco, California, USA: Morgan Kaufmann Publishers Inc., July 13, 1999, pp. 1021–1026. (Visited on Mar. 7, 2022) (cited on pages 11, 31).

[26] Michael Orlov and Moshe Sipper. 'Flight of the FINCH Through the Java Wilderness'. In: *IEEE Transactions on Evolutionary Computation* 15.2 (Apr. 2011), pp. 166–182. DOI: 10.1109/TEVC.2010.2052622 (cited on pages 11, 17, 18, 31, 83, 151, 175).

[27] Daniel Manrique, Fernando Márquez, Juan Ríos, and Alfonso Rodríguez-Patón. 'Grammar Based Crossover Operator in Genetic Programming'. In: *Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach*. Ed. by José Mira and José R. Álvarez. Lecture Notes in Computer Science. Berlin, Germany: Springer, 2005, pp. 252–261. DOI: 10.1007/11499305_26 (cited on pages 11, 31).

[28] Daniel Manrique, Juan Ríos, and Alfonso Rodríguez-Patón. *Grammar-Guided Genetic Programming*. Encyclopedia of Artificial Intelligence. 2009 (cited on pages 11, 31).

[29] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. 'Genetic Improvement of Software: A Comprehensive Survey'. In: *IEEE Transactions on Evolutionary Computation* 22.3 (June 2018), pp. 415–432. DOI: 10.1109/TEVC.2017.2693219 (cited on pages 11, 17–19, 141, 149, 150).

[30] Mohammed Javeed Zaki. 'Efficiently Mining Frequent Embedded Unordered Trees'. In: *Fundamenta Informaticae* 66.1–2 (Nov. 2004). https://dl.acm.org/doi/10.5555/1227174.1227177, pp. 33–52. (Visited on Mar. 7, 2022) (cited on pages 11–13, 48, 55, 58, 143, 147, 148).

[31] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. 'Mining API Patterns as Partial Orders from Source Code: From Usage Scenarios to Specifications'. In: *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC/FSE '07. New York, NY, USA: ACM, Sept. 7, 2007, pp. 25–34. DOI: 10.1145/1287624.1287630 (cited on pages 11, 12, 35, 47, 48, 143, 144, 147).

[32] Rakesh Agrawal and Ramakrishnan Srikant. 'Fast Algorithms for Mining Association Rules in Large Databases'. In: *Proceedings of the 20th International Conference on Very Large Data Bases*. VLDB '94. https://dl.acm.org/doi/10.5555/645920.672836. San Francisco, California, USA: Morgan Kaufmann Publishers Inc., Sept. 12, 1994, pp. 487–499. (Visited on Mar. 7, 2022) (cited on pages 11, 12, 58, 146, 147).

[33] Tatsuya Asai, Kenji Abe, Shinji Kawasoe, Hiroshi Sakamoto, Hiroki Arimura, and Setsuo Arikawa. 'Efficient Substructure Discovery from Large Semi-Structured Data'. In: *IEICE Transactions on Information and Systems* E87-D.12 (Dec. 1, 2004). https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.215.2233&rep=rep1&type=pdf, pp. 2754–2763. (Visited on Mar. 7, 2022) (cited on pages 11, 147).

[34] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. 'Mining Succinct and High-Coverage API Usage Patterns from Source Code'. In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. 2013 10th IEEE Working Conference on Mining Software Repositories (MSR 2013). San Francisco, California, USA: IEEE, May 2013, pp. 319–328. DOI: 10.1109/MSR.2013.6624045 (cited on pages 11, 12, 47, 48, 143, 145, 147).

[35] Marisa Thoma, Hong Cheng, Arthur Gretton, Jiawei Han, Hans-Peter Kriegel, Alex Smola, Le Song, Philip S. Yu, Xifeng Yan, and Karsten M. Borgwardt. 'Discriminative Frequent Subgraph Mining with Optimality Guarantees'. In: *Statistical Analysis and Data Mining: The ASA Data Science Journal* 3.5 (2010), pp. 302–318. DOI: 10.1002/sam.10084 (cited on pages 11–13, 66, 74, 75, 147, 148).

[36] Xifeng Yan, Hong Cheng, Jiawei Han, and Philip S. Yu. 'Mining Significant Graph Patterns by Leap Search'. In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD '08. New York, NY, USA: ACM, June 9, 2008, pp. 433–444. DOI: 10.1145/1376616.1376662 (cited on pages 11, 12, 48, 77, 145, 147, 148).

[37] Hong Cheng, David Lo, Yang Zhou, Xiaoyin Wang, and Xifeng Yan. 'Identifying Bug Signatures Using Discriminative Graph Mining'. In: *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*. ISSTA '09. New York, NY, USA: ACM, July 19, 2009, pp. 141–152. DOI: 10.1145/1572272.1572290 (cited on pages 12, 47, 48, 66, 74, 75, 143, 145, 147, 159).

[38] Christopher Oßner and Klemens Böhm. 'Graphs for Mining-Based Defect Localization in Multithreaded Programs'. In: *International Journal of Parallel Programming* 41.4 (Aug. 1, 2013), pp. 570–593. DOI: 10.1007/s10766-012-0237-2 (cited on pages 12, 47, 48, 143, 147).

[39] Tim A. D. Henderson and Andy Podgurski. 'Behavioral Fault Localization by Sampling Suspicious Dynamic Control Flow Subgraphs'. In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST). Apr. 2018, pp. 93–104. DOI: 10.1109/ICST.2018.00019 (cited on pages 12, 13, 68, 77, 143, 147, 148).

[40] Wei Qu, Yuanyuan Jia, and Michael Jiang. 'Pattern Mining of Cloned Codes in Software Systems'. In: *Information Sciences* 259 (Feb. 20, 2014), pp. 544–554. DOI: 10.1016/j.ins.2010.04.022 (cited on pages 12, 47, 48, 58, 66, 75, 143, 144, 147).

[41] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 'Aroma: Code Recommendation via Structural Code Search'. In: *Proceedings of the ACM on Programming Languages* 3 (OOPSLA Oct. 10, 2019), 152:1–152:28. DOI: 10.1145/3360578 (cited on pages 12, 13, 47, 48, 143–145, 147).

[42] Takashi Ishio, Hironori Date, Tatsuya Miyake, and Katsuro Inoue. 'Mining Coding Patterns to Detect Crosscutting Concerns in Java Programs'. In: *2008 15th Working Conference on Reverse Engineering*. 2008 15th Working Conference on Reverse Engineering (WCRE). Antwerp, Belgium: IEEE, Oct. 2008, pp. 123–132. DOI: 10.1109/WCRE.2008.28 (cited on pages 12, 47, 48, 143, 145, 147, 159).

[43] Yuki Ueda, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. 'Mining Source Code Improvement Patterns from Similar Code Review Works'. In: *2019 IEEE 13th International Workshop on Software Clones (IWSC)*. 2019 IEEE 13th International Workshop on Software Clones (IWSC). Hangzhou, China: IEEE, Feb. 2019, pp. 13–19. DOI: 10.1109/IWSC.2019.8665852 (cited on pages 12, 13, 47, 48, 143, 145, 147, 159).

[44] Hoan Anh Nguyen, Tien N. Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. 'Graph-Based Mining of in-the-Wild, Fine-Grained, Semantic Code Change Patterns'. In: *Proceedings of the 41st International Conference on Software Engineering*. ICSE '19. Montreal, Quebec, Canada: IEEE, May 25, 2019, pp. 819–830. DOI: 10.1109/ICSE.2019.00089 (cited on pages 12, 14, 47, 48, 66, 142, 143, 147).

[45] Chao Liu, Xifeng Yan, Hwanjo Yu, Jiawei Han, and Philip S. Yu. 'Mining Behavior Graphs for "Backtrace" of Noncrashing Bugs'. In: *Proceedings of the 2005 SIAM International Conference on Data Mining*. Proceedings of the 2005 SIAM International Conference on Data Mining. Society for Industrial and Applied Mathematics, Apr. 21, 2005, pp. 286–297. DOI: 10.1137/1.9781611972757.26 (cited on pages 12, 47, 48, 143, 146, 147).

[46] Benjamin Livshits and Thomas Zimmermann. 'DynaMine: Finding Common Error Patterns by Mining Software Revision Histories'. In: *ACM SIGSOFT Software Engineering Notes* 30.5 (Sept. 2005), pp. 296–305. DOI: 10.1145/1095430.1081754 (cited on pages 12, 47, 48, 143, 145, 147, 159).

[47] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 'Graph-Based Mining of Multiple Object Usage Patterns'. In: *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC/FSE '09. New York, NY, USA: ACM, Aug. 24, 2009, pp. 383–392. DOI: 10.1145/1595696.1595767 (cited on pages 12, 48, 142, 143, 147, 148).

[48] Anh Tuan Nguyen and Tien N. Nguyen. 'Graph-Based Statistical Language Model for Code'. In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE '15. https://dl.acm.org/doi/10.5555/2818754.2818858. Florence, Italy: IEEE, May 16, 2015, pp. 858–868. (Visited on Mar. 7, 2022) (cited on pages 12, 48, 54, 56, 57, 142, 143, 147).

[49] Quinn Hanam, Fernando S. de M. Brito, and Ali Mesbah. 'Discovering Bug Patterns in JavaScript'. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*. The 2016 24th ACM SIGSOFT International Symposium. Seattle, Washington, USA: ACM, 2016, pp. 144–156. DOI: 10.1145/2950290.2950308 (cited on pages 12, 48, 66, 143, 145–147, 159).

[50] Zsolt Balanyi and Rudolf Ferenc. 'Mining Design Patterns from C++ Source Code'. In: *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings. Sept. 2003, pp. 305–314. DOI: 10.1109/ICSM.2003.1235436 (cited on pages 12, 48, 142, 143, 147, 159).

[51] Dénes Bán and Rudolf Ferenc. 'Recognizing Antipatterns and Analyzing Their Effects on Software Maintainability'. In: *Computational Science and Its Applications – ICCSA 2014*. Ed. by Beniamino Murgante, Sanjay Misra, Ana Maria A. C. Rocha, Carmelo Torre, Jorge Gustavo Rocha, Maria Irene Falcão, David Taniar, Bernady O. Apduhan, and Osvaldo Gervasi. Lecture Notes in Computer Science. Cham, Switzerland: Springer, 2014, pp. 337–352. DOI: 10.1007/978-3-319-09156-3_25 (cited on pages 12, 48, 143, 147).

[52] Jiawei Han, Jian Pei, Behzad Mortazavi-Asl, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. 'FreeSpan: Frequent Pattern-Projected Sequential Pattern Mining'. In: *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '00. New York, NY, USA: ACM, Aug. 1, 2000, pp. 355–359. DOI: 10.1145/347090.347167 (cited on pages 12, 147, 148).

[53] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 'A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise'. In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. KDD'96. https://dl.acm.org/doi/10.5555/3001460.3001507. Portland, Oregon, USA: AAAI Press, Aug. 2, 1996, pp. 226–231. (Visited on Mar. 7, 2022) (cited on pages 12, 13, 146, 147).

[54] *OpenJDK: Graal*. URL: http://openjdk.java.net/projects/graal/ (visited on Mar. 7, 2022) (cited on page 13).

[55] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 'Partial Escape Analysis and Scalar Replacement for Java'. In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '14. New York, NY, USA: ACM, Feb. 15, 2014, pp. 165–174. DOI: 10.1145/2581122.2544157 (cited on page 13).

[56] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. 'Graal IR: An Extensible Declarative Intermediate Representation'. In: *2nd Asia-Pacific Programming Languages and Compilers Workshop (APPLC'13), as Part of the 10th Annual International Symposium on Code Generation and Optimization*. 2013, p. 9 (cited on page 13).

[57] Doug Simon, Christian Wimmer, Bernhard Urban, Gilles Duboscq, Lukas Stadler, and Thomas Würthinger. 'Snippets: Taking the High Road to a Low Level'. In: *ACM Transactions on Architecture and Code Optimization* 12.2 (June 24, 2015), 20:20:1–20:20:25. DOI: 10.1145/2764907 (cited on pages 13, 14, 174).

[58] David Leopoldseder, Lukas Stadler, Manuel Rigger, Thomas Würthinger, and Hanspeter Mössenböck. 'A Cost Model for a Graph-Based Intermediate-Representation in a Dynamic Compiler'. In: *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. VMIL '18. New York, NY, USA: ACM, Nov. 4, 2018, pp. 26–35. DOI: 10.1145/3281287.3281290 (cited on pages 13, 14, 45, 138).

[59] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, and Thomas Würthinger. 'Compilation Queuing and Graph Caching for Dynamic Compilers'. In: *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages*. VMIL '12. New York, NY, USA: ACM, Oct. 21, 2012, pp. 49–58. DOI: 10.1145/2414740.2414750 (cited on pages 14, 138, 174).

[60] Christian Wimmer and Thomas Würthinger. 'Truffle: A Self-optimizing Runtime System'. In: *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*. Tucson, Arizona, USA: ACM, 2012, pp. 13–14. DOI: 10.1145/2384716.2384723 (cited on pages 14, 49, 167).

[61] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 'One VM to Rule Them All'. In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. New York, NY, USA: ACM, Oct. 29, 2013, pp. 187–204. DOI: 10.1145/2509578.2509581 (cited on pages 14, 49, 154).

[62] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. 'High-Performance Cross-Language Interoperability in a Multi-Language Runtime'. In: *Proceedings of the 11th Symposium on Dynamic Languages*. SPLASH '15: Conference on Systems, Programming, Languages, and Applications: Software for Humanity. Pittsburgh, Pennsylvania, USA: ACM, Oct. 21, 2015, pp. 78–90. DOI: 10.1145/2816707.2816714 (cited on pages 14, 138, 154).

[63] William B. Langdon and Mark Harman. 'Optimizing Existing Software With Genetic Programming'. In: *IEEE Transactions on Evolutionary Computation* 19.1 (Feb. 2014), pp. 118–135. DOI: 10.1109/TEVC.2013.2281544 (cited on pages 17, 18).

[64] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 'Evaluating and Improving Fault Localization'. In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). May 2017, pp. 609–620. DOI: 10.1109/ICSE.2017.62 (cited on pages 17, 35, 77, 84).

[65] Oliver Krauss. 'Towards a Framework for Stochastic Performance Optimizations in Compilers and Interpreters: An Architecture Overview'. In: *Proceedings of the 15th International Conference on Managed Languages & Runtimes*. ManLang '18. New York, NY, USA: ACM, 2018, 9:1–9:7. DOI: 10.1145/3237009.3237024 (cited on pages 18, 170).

[66] Nicholas Freitag McPhee, Brian Ohs, and Tyler Hutchison. 'Semantic Building Blocks in Genetic Programming'. In: *Genetic Programming*. Ed. by Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, Anna Isabel Esparcia Alcázar, Ivanoe De Falco, Antonio Della Cioppa, and Ernesto Tarantino. Lecture Notes in Computer Science. Berlin, Germany: Springer, 2008, pp. 134–145. DOI: 10.1007/978-3-540-78671-9_12 (cited on page 19).

[67] Alberto Moraglio, Krzysztof Krawiec, and Colin G. Johnson. 'Geometric Semantic Genetic Programming'. In: *Parallel Problem Solving from Nature - PPSN XII*. Ed. by Carlos A. Coello Coello, Vincenzo Cutello, Kalyanmoy Deb, Stephanie Forrest, Giuseppe Nicosia, and Mario Pavone. Lecture Notes in Computer Science. Berlin, Germany: Springer, 2012, pp. 21–31. DOI: 10.1007/978-3-642-32937-1_3 (cited on page 19).

[68] Leonardo Vanneschi, Mauro Castelli, and Sara Silva. 'A Survey of Semantic Methods in Genetic Programming'. In: *Genetic Programming and Evolvable Machines* 15.2 (June 1, 2014), pp. 195–214. DOI: 10.1007/s10710-013-9210-0 (cited on pages 19, 151).

[69] Eric Schkufza, Rahul Sharma, and Alex Aiken. 'Stochastic Superoptimization'. In: *ACM SIGARCH Computer Architecture News* 41.1 (Mar. 16, 2013), pp. 305–316. DOI: 10.1145/2490301.2451150 (cited on page 19).

[70] David R. White. 'GI in No Time'. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '17. New York, NY, USA: ACM, July 15, 2017, pp. 1549–1550. DOI: 10.1145/3067695.3082515 (cited on pages 19, 104, 138, 150, 151).

[71] Naums Mogers, Valentin Radu, Lu Li, Jack Turner, Michael F.P. O'Boyle, and Christophe Dubach. 'Automatic Generation of Specialized Direct Convolutions for Mobile GPUs'. In: *Proceedings of the 13th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*. GPGPU '20. New York, NY, USA: ACM, Feb. 23, 2020, pp. 41–50. DOI: 10.1145/3366428.3380771 (cited on page 19).

[72] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 'Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior'. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. New York, NY, USA: ACM, Nov. 3, 2013, pp. 260–275. DOI: 10.1145/2517349.2522728 (cited on page 19).

[73] Oliver Krauss, Hanspeter Mössenböck, and Michael Affenzeller. 'Dynamic Fitness Functions for Genetic Improvement in Compilers and Interpreters'. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. The Genetic and Evolutionary Comutation Conference (GECCO). Kyoto, Japan: ACM, 2018. DOI: 10.1145/3205651.3208308 (cited on pages 20, 25, 28, 29).

[74] Hong Zhu, Patrick A. V. Hall, and John H. R. May. 'Software Unit Test Coverage and Adequacy'. In: *ACM Computing Surveys* 29.4 (Dec. 1, 1997), pp. 366–427. DOI: 10.1145/267580.267590 (cited on pages 21, 24).

[75] Laura Inozemtseva and Reid Holmes. 'Coverage Is Not Strongly Correlated with Test Suite Effectiveness'. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. New York, NY, USA: ACM, May 31, 2014, pp. 435–445. DOI: 10.1145/2568225.2568271 (cited on pages 21, 24).

[76] Andrea Arcuri and Xin Yao. 'A Novel Co-Evolutionary Approach to Automatic Software Bug Fixing'. In: *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*. 2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence). June 2008, pp. 162–168. DOI: 10.1109/CEC.2008.4630793 (cited on pages 24, 25).

[77] Gregory S. Hornby. 'ALPS: The Age-Layered Population Structure for Reducing the Problem of Premature Convergence'. In: *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*. GECCO '06. New York, NY, USA: ACM, July 8, 2006, pp. 815–822. DOI: 10.1145/1143997.1144142 (cited on page 28).

[78] Darrell Whitley, Soraya Rana, and Robert Heckendorn. 'The Island Model Genetic Algorithm: On Separability, Population Size and Convergence'. In: *Journal of Computing and Information Technology* 7 (Dec. 5, 1998). http://cit.fer.hr/index.php/CIT/article/viewFile/2919/1783. (Visited on Mar. 7, 2022) (cited on page 29).

[79] Nguyen Xuan Hoai and Robert I. McKay. *A Framework For Tree-Adjunct Grammar Guided Genetic Programming*. 2001. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.79.4037&rep=rep1&type=pdf (visited on Mar. 7, 2022) (cited on page 31).

[80] Nguyen Xuan Hoai, Robert I. McKay, Daryl Essam, and R. Chau. 'Solving the Symbolic Regression Problem with Tree-Adjunct Grammar Guided Genetic Programming: The Comparative Results'. In: *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*. Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600). Vol. 2. May 2002, 1326–1331 vol.2. DOI: 10.1109/CEC.2002.1004435 (cited on page 31).

[81] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 'Virtual Machine Warmup Blows Hot and Cold'. In: *Proceedings of the ACM on Programming Languages* 1 (OOPSLA Oct. 12, 2017), 52:1–52:27. DOI: 10.1145/3133876 (cited on pages 44, 172, 174).

[82] Michael Kommenda. *Local Optimization and Complexity Control for Symbolic Regression / Eingereicht von Michael Kommenda*. http://epub.jku.at/obvulihs/2581907. Universität Linz, 2018. (Visited on Mar. 7, 2022) (cited on page 44).

[83] Christophe Dubach, John Cavazos, Björn Franke, Grigori Fursin, Michael F.P. O'Boyle, and Olivier Temam. 'Fast Compiler Optimisation Evaluation Using Code-Feature Based Performance Prediction'. In: *Proceedings of the 4th International Conference on Computing Frontiers*. CF '07. New York, NY, USA: ACM, May 7, 2007, pp. 131–142. DOI: 10.1145/1242531.1242553 (cited on page 45).

[84] ISO/IEC 9899:2018. *Information technology - Programming languages - C*. ISO. URL: https://www.iso.org/standard/74528.html (visited on Mar. 7, 2022) (cited on pages 52, 167).

[85] Lucia Lucia, David Lo, Lingxiao Jiang, Ferdian Thung, and Aditya Budi. 'Extended Comprehensive Study of Association Measures for Fault Localization'. In: *Journal of Software: Evolution and Process* 26.2 (2014), pp. 172–219. DOI: 10.1002/smr.1616 (cited on pages 66, 68).

[86] Ning Jin, Calvin Young, and Wei Wang. 'Graph Classification Based on Pattern Co-Occurrence'. In: *CIKM*. 2009. DOI: 10.1145/1645953.1646027 (cited on page 75).

[87] Hong Cheng, Xifeng Yan, and Jiawei Han. 'Mining Graph Patterns'. In: *Managing and Mining Graph Data*. Ed. by Charu C. Aggarwal and Haixun Wang. Advances in Database Systems. Boston, Massachusetts, USA: Springer US, 2010, pp. 365–392. DOI: 10.1007/978-1-4419-6045-0_12 (cited on pages 77, 146).

[88] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. 'Formalizing Refactorings with Graph Transformations'. In: *Journal of Software Maintenance and Evolution: Research and Practice* 17.4 (2005), pp. 247–276. DOI: 10.1002/smr.316 (cited on page 83).

[89] Paolo Bottoni, Francesco Parisi Presicce, and Gabriele Taentzer. 'Specifying Integrated Refactoring with Distributed Graph Transformations'. In: *Applications of Graph Transformations with Industrial Relevance*. Ed. by John L. Pfaltz, Manfred Nagl, and Boris Böhlen. Lecture Notes in Computer Science. Berlin, Germany: Springer, 2004, pp. 220–235. DOI: 10.1007/978-3-540-25959-6_16 (cited on page 83).

[90] Frank Drewes, Berthold Hoffmann, Dirk Janssens, Mark Minas, and Niels Van Eetvelde. 'Shaped Generic Graph Transformation'. In: *Applications of Graph Transformations with Industrial Relevance*. Ed. by Andy Schürr, Manfred Nagl, and Albert Zündorf. Lecture Notes in Computer Science. Berlin, Germany: Springer, 2008, pp. 201–216. DOI: 10.1007/978-3-540-89020-1_15 (cited on page 83).

[91] Aida Radu and Sarah Nadi. 'A Dataset of Non-Functional Bugs'. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). May 2019, pp. 399–403. DOI: 10.1109/MSR.2019.00066 (cited on page 84).

[92] Chadd C. Williams and Jeffrey K. Hollingsworth. 'Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques'. In: *IEEE Transactions on Software Engineering* 31.6 (June 2005), pp. 466–480. DOI: 10.1109/TSE.2005.63 (cited on page 84).

[93] Larissa Rocha Soares, Pasqualina Potena, Ivan Do Carmo Machado, Ivica Crnkovic, and Eduardo Santana de Almeida. 'Analysis of Non-Functional Properties in Software Product Lines: A Systematic Review'. In: *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*. 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications. Aug. 2014, pp. 328–335. DOI: 10.1109/SEAA.2014.48 (cited on page 84).

[94] Fan Long and Martin Rinard. 'Automatic Patch Generation by Learning Correct Code'. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. New York, NY, USA: ACM, Jan. 11, 2016, pp. 298–312. DOI: 10.1145/2837614.2837617 (cited on page 84).

[95] Steven Gustafson. 'An Analysis of Diversity in Genetic Programming'. https://www.researchgate.net/publication/37245549_An_Analysis_of_Diversity_in_Genetic_Programming. PhD thesis. Nottingham, United Kingdom: The University of Nottingham, May 23, 2004. (Visited on Mar. 7, 2022) (cited on page 86).

[96] Edmund K. Burke, Steven Gustafson, and Graham Kendall. 'Diversity in Genetic Programming: An Analysis of Measures and Correlation with Fitness'. In: *IEEE Transactions on Evolutionary Computation* 8.1 (Feb. 2004), pp. 47–62. DOI: 10.1109/TEVC.2003.819263 (cited on page 86).

[97] Anikó Ekárt and Sándor Zoltán Németh. 'A Metric for Genetic Programs and Fitness Sharing'. In: *Genetic Programming*. Ed. by Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian Miller, Peter Nordin, and Terence C. Fogarty. Lecture Notes in Computer Science. Berlin, Germany: Springer, 2000, pp. 259–270. DOI: 10.1007/978-3-540-46239-2_19 (cited on page 86).

[98] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Third. New York, NY, USA: Cambridge University Press, 2007 (cited on pages 95, 96).

[99] William B. Langdon and Justyna Petke. 'Evolving Better Software Parameters'. en. In: *Search-Based Software Engineering*. Ed. by Thelma Elita Colanzi and Phil McMinn. Lecture Notes in Computer Science. Cham, Switzerland: Springer, 2018, pp. 363–369. DOI: 10.1007/978-3-319-99241-9_22 (cited on page 95).

[100] William B. Langdon and Justyna Petke. 'Genetic Improvement of Data Gives Binary Logarithm from Sqrt'. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '19. New York, NY, USA: ACM, July 2019, pp. 413–414. DOI: 10.1145/3319619.3321954 (cited on page 95).

[101] William B. Langdon. 'Genetic Improvement of Data Gives Double Precision Invsqrt'. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '19. New York, NY, USA: ACM, July 2019, pp. 1709–1714. DOI: 10.1145/3319619.3326800 (cited on page 95).

[102] William B. Langdon and Oliver Krauss. 'Evolving Sqrt into $1/x$ via Software Data Maintenance'. In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*. GECCO '20. New York, NY, USA: ACM, July 2020, pp. 1928–1936. DOI: 10.1145/3377929.3398110 (cited on pages 95, 150).

[103] William B. Langdon and Oliver Krauss. 'Genetic Improvement of Data for Maths Functions'. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. New York, NY, USA: ACM, 2021, pp. 31–32. DOI: 10.1145/3449726.3462730 (cited on pages 95, 134).

[104] Oliver Krauss and William B. Langdon. 'Automatically Evolving Lookup Tables for Function Approximation'. en. In: *Genetic Programming*. Ed. by Ting Hu, Nuno Lourenço, Eric Medvet, and Federico Divina. Lecture Notes in Computer Science. Cham, Switzerland: Springer, 2020, pp. 84–100. DOI: 10.1007/978-3-030-44094-7_6 (cited on pages 95, 96, 134).

[105] William B. Langdon and Oliver Krauss. 'Genetic Improvement of Data for Maths Functions'. In: *ACM Transactions on Evolutionary Learning and Optimization* 1.2 (July 2021), 7:1–7:30. DOI: 10.1145/3461016 (cited on page 95).

[106] Kinnear Kinnear. 'Generality and Difficulty in Genetic Programming: Evolving a Sort'. en. In: *Proceedings of the Fifth International Conference on Genetic Algorithms* (1993). https://citeseerx.ist.psu.edu/viewdoc/download?do (Visited on Mar. 7, 2022) (cited on pages 95, 151).

[107] Gabin An, Jinhan Kim, Seongmin Lee, and Shin Yoo. 'PyGGI: Python General framework for Genetic Improvement'. In: *Proceedings of Korea Software Congress*. KSC 2017. https://coinse.kaist.ac.kr/publications/pdfs/An2017aa.pdf. Busan, South Korea, 20-22 12 2017, pp. 536–538. (Visited on Mar. 7, 2022) (cited on pages 104, 138, 150, 151).

[108] Gabin An, Aymeric Blot, Justyna Petke, and Shin Yoo. 'PyGGI 2.0: Language Independent Genetic Improvement Framework'. en. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Tallinn, Estonia: ACM, Aug. 2019, pp. 1100–1104. DOI: 10.1145/3338906.3341184 (cited on pages 104, 138, 150, 151).

[109] Alexander E. I. Brownlee, Justyna Petke, Brad Alexander, Earl T. Barr, Markus Wagner, and David R. White. 'Gin: Genetic Improvement Research Made Easy'. en. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. Prague, Czech Republic: ACM, July 2019, pp. 985–993. DOI: 10.1145/3321707.3321841 (cited on pages 104, 138, 150, 151).

[110] Mingyi Lim, Giovani Guizzo, and Justyna Petke. 'Impact of Test Suite Coverage on Overfitting in Genetic Improvement of Software'. In: *Search-Based Software Engineering*. Ed. by Aldeida Aleti and Annibale Panichella. Vol. 12420. Lecture Notes in Computer Science. Cham, Switzerland: Springer, 2020, pp. 188–203. DOI: 10.1007/978-3-030-59762-7_14 (cited on page 128).

[111] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 'Automatic Patch Generation Learned from Human-Written Patches'. In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. https://dl.acm.org/doi/10.5555/2486788.2486893. San Francisco, California, USA: IEEE, May 2013, pp. 802–811. (Visited on Mar. 7, 2022) (cited on pages 133, 152).

[112] Alexander E. I. Brownlee, Justyna Petke, and Anna F. Rasburn. 'Injecting Shortcuts for Faster Running Java Code'. In: *2020 IEEE Congress on Evolutionary Computation (CEC)*. 2020 IEEE Congress on Evolutionary Computation (CEC). Glasgow, United Kingdom: IEEE, July 2020, pp. 1–8. DOI: 10.1109/CEC48606.2020.9185708 (cited on pages 138, 155, 174).

[113] Wouter Poncin, Alexander Serebrenik, and Mark van den Brand. 'Process Mining Software Repositories'. In: *2011 15th European Conference on Software Maintenance and Reengineering*. 2011 15th European Conference on Software Maintenance and Reengineering. Mar. 2011, pp. 5–14. DOI: 10.1109/CSMR.2011.5 (cited on page 142).

[114] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 'Learning Natural Coding Conventions'. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. New York, NY, USA: ACM, Nov. 11, 2014, pp. 281–293. DOI: 10.1145/2635868.2635883 (cited on page 142).

[115] Stas Negara, Mihai Codoban, Danny Dig, and Ralph E. Johnson. 'Mining Fine-Grained Code Changes to Detect Unknown Change Patterns'. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE '14. New York, NY, USA: ACM, May 31, 2014, pp. 803–813. DOI: 10.1145/2568225.2568317 (cited on page 142).

[116] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. 'Comparing Approaches to Mining Source Code for Call-Usage Patterns'. In: *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*. Fourth International Workshop on Mining Software Repositories. Minneapolis, MN, USA: IEEE, May 2007, pp. 20–20. DOI: 10.1109/MSR.2007.3 (cited on page 142).

[117] Etem Deniz and Alper Sen. 'Using Machine Learning Techniques to Detect Parallel Patterns of Multi-threaded Applications'. In: *International Journal of Parallel Programming* 44.4 (Aug. 1, 2016), pp. 867–900. DOI: 10.1007/s10766-015-0396-z (cited on page 142).

[118] Song Wang, Taiyue Liu, and Lin Tan. 'Automatically Learning Semantic Features for Defect Prediction'. In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). May 2016, pp. 297–308. DOI: 10.1145/2884781.2884804 (cited on page 142).

[119] László Vidács, Martin Gogolla, and Rudolf Ferenc. 'From C++ Refactorings to Graph Transformations'. In: *Electronic Communications of the EASST* 3.0 (0 Feb. 20, 2007). DOI: 10.14279/tuj.eceasst.3.10 (cited on page 142).

[120] Zoltán Ujhelyi, Ákos Horváth, Dániel Varró, Norbert István Csiszár, Gábor Szőke, László Vidács, and Rudolf Ferenc. 'Anti-Pattern Detection with Model Queries: A Comparison of Approaches'. In: *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE). Feb. 2014, pp. 293–302. DOI: 10.1109/CSMR-WCRE.2014.6747181 (cited on page 143).

[121] Zoltán Ujhelyi, Gábor Szőke, Ákos Horváth, Norbert István Csiszár, László Vidács, Dániel Varró, and Rudolf Ferenc. 'Performance Comparison of Query-Based Techniques for Anti-Pattern Detection'. In: *Information and Software Technology* 65 (Sept. 1, 2015), pp. 147–165. DOI: 10.1016/j.infsof.2015.01.003 (cited on page 143).

[122] Jim Gray. *Why Do Computers Stop And What Can Be Done About It?* https://www.hpl.hp.com/techreports/tandem/TR-85.7.pdf. 1985. (Visited on Mar. 7, 2022) (cited on page 144).

[123] Chuntao Jiang, Frans Coenen, and Michele Zito. 'A Survey of Frequent Subgraph Mining Algorithms'. In: *The Knowledge Engineering Review* 28.1 (Mar. 2013), pp. 75–105. DOI: 10.1017/S0269888912000331 (cited on page 146).

[124] Mohammed Javeed Zaki. 'Efficiently Mining Frequent Trees in a Forest'. In: *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '02. New York, NY, USA: ACM, July 23, 2002, pp. 71–80. DOI: 10.1145/775047.775058 (cited on pages 147, 148).

[125] Mohammed Javeed Zaki. 'Efficiently Mining Frequent Trees in a Forest: Algorithms and Applications'. In: *IEEE Transactions on Knowledge and Data Engineering* 17.8 (2005), pp. 1021–1035. DOI: 10.1109/TKDE.2005.125 (cited on pages 147, 148).

[126] Xifeng Yan and Jiawei Han. 'gSpan: Graph-Based Substructure Pattern Mining'. In: *2002 IEEE International Conference on Data Mining, 2002. Proceedings.* 2002 IEEE International Conference on Data Mining, 2002. Proceedings. Dec. 2002, pp. 721–724. DOI: 10.1109/ICDM.2002.1184038 (cited on pages 147, 148).

[127] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. 'PrefixSpan,: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth'. In: *Proceedings 17th International Conference on Data Engineering*. Proceedings 17th International Conference on Data Engineering. Apr. 2001, pp. 215–224. DOI: 10.1109/ICDE.2001.914830 (cited on pages 147, 148).

[128] Jianyong Wang and Jiawei Han. 'BIDE: Efficient Mining of Frequent Closed Sequences'. In: *Proceedings. 20th International Conference on Data Engineering*. Proceedings 20th International Conference on Data Engineering. Apr. 2004, pp. 79–90. doi: 10.1109/ICDE.2004.1319986 (cited on pages 147, 148).

[129] Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. 'An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data'. In: *Principles of Data Mining and Knowledge Discovery*. Ed. by Djamel Abdelkader Zighed, Jan Komorowski, and Jan Żytkow. Lecture Notes in Computer Science. Berlin, Germany: Springer, 2000, pp. 13–23. doi: 10.1007/3-540-45372-5_2 (cited on page 146).

[130] Hoang Son Pham, Siegfried Nijssen, Kim Mens, Dario Di Nucci, Tim Molderez, Coen De Roover, Johan Fabry, and Vadim Zaytsev. 'Mining Patterns in Source Code Using Tree Mining Algorithms'. In: *Discovery Science*. Ed. by Petra Kralj Novak, Tomislav Šmuc, and Sašo Džeroski. Lecture Notes in Computer Science. Cham, Switzerland: Springer, 2019, pp. 471–480. doi: 10.1007/978-3-030-33778-0_35 (cited on page 147).

[131] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. 'Test-Driven Synthesis'. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. New York, NY, USA: ACM, June 2014, pp. 408–418. doi: 10.1145/2594291.2594297 (cited on page 149).

[132] Sumit Gulwani. 'Synthesis from Examples: Interaction Models and Algorithms'. In: *2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. Sept. 2012, pp. 8–14. doi: 10.1109/SYNASC.2012.69 (cited on page 149).

[133] Xinyun Chen, Chang Liu, and Dawn Song. 'Tree-to-Tree Neural Networks for Program Translation'. In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS'18. https://dl.acm.org/doi/10.5555/3327144.3327180. Red Hook, NY, USA: Curran Associates Inc., Dec. 2018, pp. 2552–2562. (Visited on Mar. 7, 2022) (cited on page 149).

[134] Shin Yoo. 'Embedding Genetic Improvement into Programming Languages'. en. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. Berlin, Germany: ACM, July 2017, pp. 1551–1552. doi: 10.1145/3067695.3082516 (cited on page 150).

[135] Gabin An, Jinhan Kim, and Shin Yoo. 'Comparing Line and AST Granularity Level for Program Repair Using PʏGGI'. en. In: *Proceedings of the 4th International Workshop on Genetic Improvement Workshop*. Gothenburg, Sweden: ACM, June 2018, pp. 19–26. doi: 10.1145/3194810.3194814 (cited on page 151).

[136] Michael Kommenda, Gabriel Kronberger, Stefan Wagner, Stephan Winkler, and Michael Affenzeller. 'On the Architecture and Implementation of Tree-based Genetic Programming in HeuristicLab'. In: *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*. GECCO '12. Philadelphia, Pennsylvania, USA: ACM, 2012, pp. 101–108. doi: 10.1145/2330784.2330801 (cited on page 151).

[137] William B. Langdon. 'Genetic Improvement of Software for Multiple Objectives'. In: *Search-Based Software Engineering*. Springer, 2015. doi: 10.1007/978-3-319-22183-0_2 (cited on page 151).

[138] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 'GenProg: A Generic Method for Automatic Software Repair'. In: *IEEE Transactions on Software Engineering* 38.1 (Jan. 2012), pp. 54–72. doi: 10.1109/TSE.2011.104 (cited on page 152).

[139] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. 'Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results'. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Silicon Valley, CA, USA: IEEE, Nov. 2013, pp. 356–366. doi: 10.1109/ASE.2013.6693094 (cited on page 152).

[140] Istvan Jonyer and Akiko Himes. 'Improving Modularity in Genetic Programming Using Graph-Based Data Mining.' In: *FLAIRS Conference*. https://www.aaai.org/Papers/FLAIRS/2006/Flairs06-110.pdf. 2006, pp. 556–561. (Visited on Mar. 7, 2022) (cited on page 152).

[141] William B. Langdon and Wolfgang Banzhaf. 'Repeated Patterns in Genetic Programming'. en. In: *Natural Computing* 7.4 (Dec. 2008), pp. 589–613. doi: 10.1007/s11047-007-9038-8 (cited on page 152).

[142] William B. Langdon and Wolfgang Banzhaf. 'Repeated Patterns in Tree Genetic Programming'. In: *Genetic Programming*. Ed. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, Chandrasekaran Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Maarten Keijzer, Andrea Tettamanzi, Pierre Collet, Jano van Hemert, and Marco Tomassini. Vol. 3447. Berlin, Germany: Springer, 2005, pp. 190–202. DOI: 10.1007/978-3-540-31989-4_17 (cited on page 152).

[143] Nicholas Freitag McPhee, Brian Ohs, and Tyler Hutchison. 'Semantic Building Blocks in Genetic Programming'. In: *Genetic Programming*. Ed. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, Chandrasekaran Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, Anna Isabel Esparcia Alcázar, Ivanoe De Falco, Antonio Della Cioppa, and Ernesto Tarantino. Vol. 4971. Berlin, Germany: Springer, 2008, pp. 134–145. DOI: 10.1007/978-3-540-78671-9_12 (cited on page 152).

[144] Nicholas Freitag McPhee and Nicholas J. Hopper. 'Analysis of Genetic Diversity through Population History'. In: *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 2*. GECCO '99. https://dl.acm.org/doi/10.5555/2934046.2934071. San Francisco, California, USA: Morgan Kaufmann Publishers Inc., July 1999, pp. 1112–1120. (Visited on Mar. 7, 2022) (cited on page 153).

[145] Nicholas Freitag McPhee, Maggie M. Casale, Mitchell D. Finzel, Thomas Helmuth, and Lee Spector. 'Visualizing Genetic Programming Ancestries'. In: *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*. GECCO '16. New York, NY, USA: ACM, July 2016, pp. 1419–1426. DOI: 10.1145/2908961.2931741 (cited on page 153).

[146] Nicholas Freitag McPhee, Mitchell D. Finzel, Maggie M. Casale, Thomas Helmuth, and Lee Spector. 'A Detailed Analysis of a PushGP Run'. In: *Genetic Programming Theory and Practice XIV*. Ed. by Rick Riolo, Bill Worzel, Brian Goldman, and Bill Tozier. Cham, Switzerland: Springer, 2018, pp. 65–83. DOI: 10.1007/978-3-319-97088-2_5 (cited on page 153).

[147] Nicholas Freitag McPhee, David Donatucci, and Thomas Helmuth. 'Using Graph Databases to Explore the Dynamics of Genetic Programming Runs'. In: *Genetic Programming Theory and Practice XIII*. Ed. by Rick Riolo, William P. Worzel, Mark Kotanchek, and Arthur Kordon. Cham, Switzerland: Springer, 2016, pp. 185–201. DOI: 10.1007/978-3-319-34223-8_11 (cited on page 153).

[148] David Donatucci, Kirbie M. Dramdahl, and Nicholas Freitag McPhee. *Analysis of Genetic Programming Ancestry Using a Graph Database*. en. http://www.micsymposium.org/mics2014/ProceedingsMICS_2014/mics2014_submission_40.pdf. 2014. (Visited on Mar. 7, 2022) (cited on page 153).

[149] Nicholas Freitag McPhee, Maggie M. Casale, Mitchell D. Finzel, Thomas Helmuth, and Lee Spector. 'Visualizing Genetic Programming Ancestries Using Graph Databases'. en. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. Berlin, Germany: ACM, July 2017, pp. 245–246. DOI: 10.1145/3067695.3075617 (cited on page 153).

[150] Bogdan Burlacu, Michael Affenzeller, Michael Kommenda, Stephan Winkler, and Gabriel Kronberger. 'Visualization of Genetic Lineages and Inheritance Information in Genetic Programming'. In: *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation*. GECCO '13. New York, NY, USA: ACM, July 2013, pp. 1351–1358. DOI: 10.1145/2464576.2482714 (cited on page 153).

[151] Bogdan Burlacu, Lukas Kammerer, Michael Affenzeller, and Gabriel Kronberger. *Hash-Based Tree Similarity and Simplification in Genetic Programming for Symbolic Regression*. July 22, 2021 (cited on page 153).

[152] Justyna Petke. 'New Operators for Non-Functional Genetic Improvement'. en. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. Berlin, Germany: ACM, July 2017, pp. 1541–1542. DOI: 10.1145/3067695.3082520 (cited on page 153).

[153] James Callan, Oliver Krauss, Justyna Petke, and Federica Sarro. 'How Do Android Developers Improve Non-FunctionalProperties of Software?' In: *Empirical Software Engineering* () (cited on page 153).

[154] Brendan Cody-Kenny, Michael Fenton, and Michael O'Neill. 'From Problem Landscapes to Language Landscapes: Questions in Genetic Improvement'. en. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. Berlin, Germany: ACM, July 2017, pp. 1509–1510. DOI: 10.1145/3067695.3082522 (cited on page 153).

[155] Krzysztof Krawiec and Jerry Swan. 'Pattern-Guided Genetic Programming'. en. In: *Proceeding of the Fifteenth Annual Conference on Genetic and Evolutionary Computation Conference - GECCO '13*. Amsterdam, The Netherlands: ACM, 2013, p. 949. DOI: 10.1145/2463372.2463496 (cited on page 153).

[156] Henry Massalin. 'Superoptimizer: A Look at the Smallest Program'. In: *ACM SIGARCH Computer Architecture News* 15.5 (Oct. 1, 1987), pp. 122–126. DOI: 10.1145/36177.36194 (cited on pages 154, 155).

[157] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 'Lazy Code Motion'. In: *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*. PLDI '92. New York, NY, USA: ACM, July 1, 1992, pp. 224–234. DOI: 10.1145/143095.143136 (cited on pages 154, 157).

[158] Anjali Mahajan and Mir Sadique Ali. 'Superblock Scheduling Using Genetic Programming for Embedded Systems'. In: *2008 7th IEEE International Conference on Cognitive Informatics*. 2008 7th IEEE International Conference on Cognitive Informatics (ICCI). Stanford University, California, USA: IEEE, Aug. 2008, pp. 261–266. DOI: 10.1109/COGINF.2008.4639177 (cited on page 154).

[159] Mark Stephenson, Saman Amarasinghe, Martin C. Martin, and Una-May O'Reilly. 'Meta Optimization: Improving Compiler Heuristics with Machine Learning'. In: *ACM SIGPLAN Notices* 38.5 (May 9, 2003), pp. 77–90. DOI: 10.1145/780822.781141 (cited on page 154).

[160] Mark Stephenson, Una-May O'Reilly, Martin C. Martin, and Saman Amarasinghe. 'Genetic Programming Applied to Compiler Heuristic Optimization'. In: *Proceedings of the 6th European Conference on Genetic Programming*. EuroGP'03. https://dl.acm.org/doi/10.5555/1762668.1762691. Berlin, Germany: Springer, Apr. 14, 2003, pp. 238–253. (Visited on Mar. 7, 2022) (cited on page 154).

[161] Mark Stephenson and Saman Amarasinghe. 'Predicting Unroll Factors Using Supervised Classification'. In: *International Symposium on Code Generation and Optimization*. International Symposium on Code Generation and Optimization. Mar. 2005, pp. 123–134. DOI: 10.1109/CGO.2005.29 (cited on page 154).

[162] Martin Kristien, Bruno Bodin, Michel Steuwer, and Christophe Dubach. 'High-Level Synthesis of Functional Patterns with Lift'. In: *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*. ARRAY 2019. Phoenix, Arizona, USA: ACM, 2019, pp. 35–45. DOI: 10.1145/3315454.3329957 (cited on page 154).

[163] Naums Mogers, Michel Steuwer, Aaron Smith, Christophe Dubach, Dimitrios Vytiniotis, and Ryota Tomioka. 'Towards Mapping Lift to Deep Neural Network Accelerators'. English. In: *1st HiPEAC Workshop on Emerging Deep Learning Accelerators (EDLA)*. http://workshops.inf.ed.ac.uk/edla/. Jan. 2019. (Visited on Mar. 7, 2022) (cited on page 154).

[164] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 'Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages'. In: *ACM Transactions on Embedded Computing Systems* 13 (4s Apr. 1, 2014), 134:1–134:25. DOI: 10.1145/2584665 (cited on page 154).

[165] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 'Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines'. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. New York, NY, USA: ACM, June 16, 2013, pp. 519–530. DOI: 10.1145/2491956.2462176 (cited on page 154).

[166] Dongni Han, Shixiong Xu, Li Chen, and Lei Huang. 'PADS: A Pattern-Driven Stencil Compiler-Based Tool for Reuse of Optimizations on GPGPUs'. In: *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. 2011 IEEE 17th International Conference on Parallel and Distributed Systems. Dec. 2011, pp. 308–315. DOI: 10.1109/ICPADS.2011.94 (cited on page 154).

[167] Howard C. Lovatt, Anthony M. Sloane, and Dominic R. Verity. 'A Pattern Enforcing Compiler (PEC) for Java: Using the Compiler'. In: *Proceedings of the 2nd Asia-Pacific Conference on Conceptual Modelling - Volume 43*. APCCM '05. https://dl.acm.org/doi/10.5555/1082276.1082285. Australia: Australian Computer Society, Inc., Jan. 1, 2005, pp. 69–78. (Visited on Mar. 7, 2022) (cited on page 154).

[168] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. 'Tom: Piggybacking Rewriting on Java'. In: *Term Rewriting and Applications*. Ed. by Franz Baader. Lecture Notes in Computer Science. Berlin, Germany: Springer, 2007, pp. 36–47. DOI: 10.1007/978-3-540-73449-9_5 (cited on page 154).

[169] Grigori Fursin. 'Collective Tuning Initiative: automating and accelerating development and optimization of computing systems'. In: *Proceedings of the GCC Developers' Summit*. https://hal.inria.fr/inria-00436029v2/file/fursin.pdf. Montreal, Quebec, Canada, June 2009. (Visited on Mar. 7, 2022) (cited on page 155).

[170] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin V. Bonilla, John Thomson, Christopher K. I. Williams, and Michael O'Boyle. 'Milepost GCC: Machine Learning Enabled Self-tuning Compiler'. In: *International Journal of Parallel Programming* 39.3 (June 1, 2011), pp. 296–327. DOI: 10.1007/s10766-010-0161-2 (cited on page 155).

[171] Oliver Krauss. *Amaru - The Amaru Framework for Genetic Improvement and Pattern Mining in Graal and Truffle*. Version 1.0. Feb. 2021. DOI: 10.5281/zenodo.6104384 (cited on pages 155, 169).

[172] Grigori Fursin, Anton Lokhmotov, Dmitry Savenko, and Eben Upton. *A Collective Knowledge Workflow for Collaborative Research into Multi-Objective Autotuning and Machine Learning Techniques*. Jan. 19, 2018. URL: http://arxiv.org/abs/1801.08024 (visited on Mar. 7, 2022) (cited on page 155).

[173] Christos Kartsaklis, Oscar Hernandez, Chung-Hsing Hsu, Thomas Ilsche, Wayne Joubert, and Richard L. Graham. 'HERCULES: A Pattern Driven Code Transformation System'. In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*. 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum. May 2012, pp. 574–583. DOI: 10.1109/IPDPSW.2012.69 (cited on page 155).

[174] Eunjung Park, Christos Kartsaklis, and John Cavazos. 'HERCULES: Strong Patterns towards More Intelligent Predictive Modeling'. In: *2014 43rd International Conference on Parallel Processing*. 2014 43rd International Conference on Parallel Processing. Sept. 2014, pp. 172–181. DOI: 10.1109/ICPP.2014.26 (cited on page 155).

[175] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 'Conditionally Correct Superoptimization'. In: *ACM SIGPLAN Notices* 50.10 (Oct. 23, 2015), pp. 147–162. DOI: 10.1145/2858965.2814278 (cited on page 155).

[176] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 'Scaling up Superoptimization'. In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '16. New York, NY, USA: ACM, Mar. 25, 2016, pp. 297–310. DOI: 10.1145/2872362.2872387 (cited on page 155).

[177] Abhinav Jangda and Greta Yorsh. 'Unbounded Superoptimization'. In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2017. New York, NY, USA: ACM, Oct. 25, 2017, pp. 78–88. DOI: 10.1145/3133850.3133856 (cited on page 155).

[178] Manasij Mukherjee, Pranav Kant, Zhengyang Liu, and John Regehr. 'Dataflow-Based Pruning for Speeding up Superoptimization'. In: *Proceedings of the ACM on Programming Languages* 4 (OOPSLA Nov. 13, 2020), pp. 1–24. DOI: 10.1145/3428245 (cited on pages 155, 156).

[179] Eric Schkufza, Rahul Sharma, and Alex Aiken. 'Stochastic Superoptimization'. In: *ACM SIGPLAN Notices* 48.4 (Mar. 16, 2013), pp. 305–316. DOI: 10.1145/2499368.2451150 (cited on page 156).

[180] Berkeley Churchill, Rahul Sharma, Jf Bastien, and Alex Aiken. 'Sound Loop Superoptimization for Google Native Client'. In: *ACM SIGARCH Computer Architecture News* 45.1 (May 11, 2017), pp. 313–326. DOI: 10.1145/3093337.3037754 (cited on page 156).

[181] Rudy Bunel, Alban Desmaison, M. Pawan Kumar, Philip H. S. Torr, and Pushmeet Kohli. *Learning to Superoptimize Programs*. June 28, 2017. URL: http://arxiv.org/abs/1611.01787 (visited on Mar. 7, 2022) (cited on page 156).

[182] Alex Shypula, Pengcheng Yin, Jeremy Lacomis, Claire Le Goues, Edward Schwartz, and Graham Neubig. *Learning to Superoptimize Real-world Programs*. Sept. 28, 2021. URL: http://arxiv.org/abs/2109.13498 (visited on Mar. 7, 2022) (cited on page 156).

[183] Junhyuk Oh, Yijie Guo, Satinder Singh, and Honglak Lee. 'Self-Imitation Learning'. In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. http://proceedings.mlr.press/v80/oh18b/oh18b.pdf. PMLR, Oct. 2018, pp. 3878–3887. (Visited on Mar. 7, 2022) (cited on page 156).

[184] Eric Schulte, Stephanie Forrest, and Westley Weimer. 'Automated Program Repair through the Evolution of Assembly Code'. In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. ASE '10. New York, NY, USA: ACM, Sept. 20, 2010, pp. 313–316. DOI: 10.1145/1858996.1859059 (cited on page 156).

[185] Vinicius Paulo L. Oliveira, Eduardo Faria de Souza, Claire Le Goues, and Celso G. Camilo-Junior. 'Improved Representation and Genetic Operators for Linear Genetic Programming for Automated Program Repair'. In: *Empirical Software Engineering* 23.5 (Oct. 1, 2018), pp. 2980–3006. DOI: 10.1007/s10664-017-9562-9 (cited on page 156).

[186] Kunal Banerjee, Chandan Karfa, Dipankar Sarkar, and Chittaranjan Mandal. 'Verification of Code Motion Techniques Using Value Propagation'. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33.8 (Aug. 2014), pp. 1180–1193. DOI: 10.1109/TCAD.2014.2314392 (cited on page 157).

[187] Zheng Wang and Michael O'Boyle. 'Machine Learning in Compiler Optimization'. In: *Proceedings of the IEEE* 106.11 (Nov. 2018), pp. 1879–1901. DOI: 10.1109/JPROC.2018.2817118 (cited on page 157).

[188] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 'A Survey on Compiler Autotuning Using Machine Learning'. In: *ACM Computing Surveys* 51.5 (Jan. 23, 2019), pp. 1–42. DOI: 10.1145/3197978 (cited on page 157).

[189] Amir H. Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. 'MiCOMP: Mitigating the Compiler Phase-Ordering Problem Using Optimization Sub-Sequences and Machine Learning'. In: *ACM Transactions on Architecture and Code Optimization* 14.3 (Sept. 6, 2017), pp. 1–28. DOI: 10.1145/3124452 (cited on page 157).

[190] Christophe Dubach, Timothy M. Jones, Edwin V. Bonilla, Grigori Fursin, and Michael F.P. O'Boyle. 'Portable Compiler Optimisation across Embedded Programs and Microarchitectures Using Machine Learning'. In: *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). Dec. 2009, pp. 78–88. DOI: 10.1145/1669112.1669124 (cited on page 157).

[191] Raphael Mosaner. 'Machine Learning to Ease Understanding of Data Driven Compiler Optimizations'. In: *Companion Proceedings of the 2020 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. New York, NY, USA: ACM, Nov. 15, 2020, pp. 4–6. DOI: 10.1145/3426430.3429451 (cited on page 158).

[192] Hanspeter Mössenböck, Albrecht Wöß, and Markus Löberbauer. *The Compiler Generator Coco/R*. https://ssw.jku.at/Research/Projects/Coco/Doc/UserManual.pdf. Citeseer, 2003. (Visited on Mar. 7, 2022) (cited on page 167).

[193] Hanspeter Mössenböck, Markus Löberbauer, Albrecht Wöß, and University of Linz. *The Compiler Generator Coco/R*. URL: https://ssw.jku.at/Research/Projects/Coco/ (visited on Mar. 7, 2022) (cited on page 167).

[194] Stefan Wagner. 'Heuristic Optimization Software Systems'. https://research.fh-ooe.at/data/files/publications/1341_wagner2009.pdf. PhD thesis. Linz, Austria: Johannes Kepler University Linz, Mar. 2009. (Visited on Mar. 7, 2022) (cited on pages 177, 178).

[195] Stefan Wagner, Stephan Winkler, Erik Pitzer, Gabriel Kronberger, Andreas Beham, Roland Braune, and Michael Affenzeller. 'Benefits of Plugin-Based Heuristic Optimization Software Systems'. In: *Computer Aided Systems Theory – EUROCAST 2007*. Ed. by Roberto Moreno Díaz, Franz Pichler, and Alexis Quesada Arencibia. Lecture Notes in Computer Science. Berlin, Germany: Springer, 2007, pp. 747–754. DOI: 10.1007/978-3-540-75867-9_94 (cited on pages 177, 178).

[196] Stefan Wagner, Gabriel Kronberger, Andreas Beham, Michael Kommenda, Andreas Scheibenpflug, Erik Pitzer, Stefan Vonolfen, Monika Kofler, Stephan Winkler, Viktoria Dorfer, and Michael Affenzeller. 'Architecture and Design of the HeuristicLab Optimization Environment'. In: *Advanced Methods and Applications in Computational Intelligence*. Ed. by Ryszard Klempous, Jan Nikodem, Witold Jacak, and Zenon Chaczko. Topics in Intelligent Engineering and Informatics. Heidelberg, Germany: Springer, 2014, pp. 197–261. DOI: 10.1007/978-3-319-01436-4_10 (cited on pages 177, 178).

[197] Michael Kommenda, Gabriel Kronberger, Stefan Wagner, Stephan Winkler, and Michael Affenzeller. 'On the Architecture and Implementation of Tree-Based Genetic Programming in HeuristicLab'. In: *Proceedings of the Fourteenth International Conference on Genetic and Evolutionary Computation Conference Companion - GECCO Companion '12*. The Genetic and Evolutionary Comutation Conference (GECCO). Philadelphia, Pennsylvania, USA: ACM, 2012, p. 101. DOI: 10.1145/2330784.2330801 (cited on pages 177, 179).

[198] Oliver Krauss and Daniel Dorfmeister. *HeuristicLab Connector - Connecting HeuristicLab to the Amaru Framework for Genetic Improvement and Pattern Mining* (cited on page 177).

[199] Daniel Dorfmeister and Oliver Krauss. 'Integrating HeuristicLab with Compilers and Interpreters for Non-Functional Code Optimization'. In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*. The Genetic and Evolutionary Comutation Conference (GECCO). Cancun, Mexico: ACM, July 2020. DOI: 10.1145/3377929.3398103 (cited on pages 177–180).

[200] Andreas Scheibenpflug, Stefan Wagner, Gabriel Kronberger, and Michael Affenzeller. 'HeuristicLab Hive - An Open Source Environment for Parallel and Distributed Execution of Heuristic Optimization Algorithms'. In: *1st Australian Conference on the Applications of Systems Engineering ACASE'12*. Ed. by Robin Braun and Zenon Chaczko. https://www.researchgate.net/profile/Anup-Kale/publication/320467448_Evolutionary_Feature_Optimization_for_Robust_Multimodal_Object_Detection/links/59e73d9caca2721fc23086c0/Evolutionary-Feature-Optimization-for-Robust-Multimodal-Object-Detection.pdf#page=63. Sydney, Australia, Feb. 2012, pp. 63–65. (Visited on Mar. 7, 2022) (cited on page 178).

List of Figures

List of Tables

List of Listings

Acronyms

A

AGM Apriori-based Graph Mining. 146

ALPS Age Layered Population Structure. 28

AST Abstract Syntax Tree. vii, viii, 3, 5, 6, 9, 12, 14, 16, 20–26, 28, 29, 31–39, 43–56, 58, 61–65, 67–70, 72, 75–79, 81–89, 91–93, 95–98, 101–111, 114–120, 122–139, 146, 149–156, 159–163, 167, 171–175, 179, 180, 183, 225–227

B

BIDE BI-Directional Extension. 148

C

CK Collective Knowledge Framework. 155

CNN Convolutional Neural Network. 19

CORK correspondence-based quality criterion. 148

D

DNN Deep Neural Network. 154

DOF Degree of Freedom. 33, 39, 42, 88, 90

DPML Design Pattern Markup Language. 142

DSL domain-specific language. 154, 177, 178

E

EBNF Extended Backus-Naur Form. 44, 167

F

FINCH Fertile Darwinian Bytecode Harvester. 11

FPGA field-programmable gate array. 154

G

GGGP Grammar Guided Genetic Programming. 11, 31

GI Genetic Improvement. v, vi, viii, 3–7, 9–11, 13, 14, 16–26, 28, 29, 31, 32, 34, 35, 39, 45, 46, 64, 76, 78, 81–86, 93, 95, 96, 104, 107, 128, 134, 138, 141, 149–157, 159–161, 163, 167–171, 174, 179

GIN GI in No Time. 19

GP Genetic Programming. viii, 6, 9, 10, 17, 19, 24, 28, 29, 31, 86, 95, 104, 138, 141, 149, 151–154, 156, 171, 177, 225

GPU graphics processing unit. 19

gSpan graph-based Substructure pattern mining. 148

GUI graphical user interface. 177, 178

H

HTML Hyper Text Markup Language. 169

I

IGOR Independent Growth of Ordered Relationships. v, vi, viii, 5, 13, 47, 68–71, 98, 99, 101, 107, 108, 130, 141, 148, 149, 159, 160, 162, 169–171

IQR interquartile range. 130

IR intermediate representation. 13, 14, 99–101, 154, 162, 225

J

JIT just-in-time. 13

JVM Java Virtual Machine. 104, 108, 112, 118, 124, 175

K

KGGI Knowledge-guided Genetic Improvement. v–viii, 6, 7, 17, 31, 32, 34, 43, 45, 46, 81–83, 86, 87, 89, 95, 96, 98, 101, 103–111, 119–129, 131, 132, 136–138, 151, 153, 157, 159–163, 168–170, 172, 173, 179, 183, 225, 226

L

LIM Language Independent Model. 143

N

NFP Non-Functional Property. 3–7, 9, 11–14, 17–19, 22, 23, 26, 31, 33, 37–39, 42, 44–49, 52, 64, 65, 75, 81, 82, 84–86, 89, 96, 107, 118, 120, 130, 146, 149, 150, 153, 154, 156, 159–162, 171, 172, 174, 175

P

PAR Pattern based Automatic program Repair. 152

PDG program dependence graph. 14, 144

S

SBSE Search-Based Software Engineering. v, 5, 10, 17, 149, 155, 156, 159

SIL Self Imitation learning. 156

SILO Self Imitation learning for Optimization. 156

SMT satisfiability modulo theory. 155, 156

SVM Support Vector Machine. 146

T

T3GP Tree-adjunct Grammar Guided Genetic Programming. 31

TGP Tree Genetic Programming. 31

X

XML extensible markup language. 142