STRENGTH AND LIMITATIONS OF REINFORCEMENT LEARNING AND MONTE CARLO METHODS FOR GENERATING PATHOLOGICAL PERFORMANCE TEST CASES

by

ZIYAD ALSAEED

A DISSERTATION

Presented to the Department of Computer and Information Science
and the Division of Graduate Studies of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

December 2021

DISSERTATION APPROVAL PAGE

Student: Ziyad Alsaeed

Title: Strength and Limitations of Reinforcement Learning and Monte Carlo Methods for Generating Pathological Performance Test Cases

This dissertation has been accepted and approved in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science by:

Michal Young	Chair
Stephen Fickas	Core Member
Christopher Wilson	Core Member
Julie Hessler	Institutional Representative

and

Krista Chronister	Vice Provost for Graduate Studies

Original approval signatures are on file with the University of Oregon Division of Graduate Studies.

Degree awarded December 2021

DISSERTATION ABSTRACT

Ziyad Alsaeed

Doctor of Philosophy

Department of Computer and Information Science

December 2021

Title: Strength and Limitations of Reinforcement Learning and Monte Carlo Methods for Generating Pathological Performance Test Cases

Detecting and repairing software performance issues requires test cases that demonstrate those problems. The quality and availability of test cases play an instrumental role in applications performance testing. Worst-case complexity edge cases often escape developers' understanding as the size and complexity of the application grow. Research shows that feedback-directed search (mutational fuzzing) can effectively discover pathological inputs that expose performance issues, but blindly mutating byte strings slows search by producing mostly invalid inputs. The search can be accelerated for applications that accept richly structured textual input by adapting search techniques with grammar-based generation. Monte Carlo tree search (MCTS, a random sampling search method) and reinforcement learning (RL, a machine learning-based technique to learn through environment interactions) are two unexplored paths in the domain of pathological input generation.

MCTS and RL have been applied to different domains, with notable success in the game domain. Adapting these techniques to search for inputs that trigger slow processing in a diverse set of applications poses different challenges. Primarily, adapting feedback-directed techniques from game search to a domain

with widely varying rewards jeopardizes the search effort by skewing toward initial and mostly trivial observations. We devise different adaptive reward functions that perform well despite the diversity in application cost ranges and grammars. Other challenges vary depending on the applied search technique (e.g., the quality of the state representation). We overcome each challenge by applying a dynamic solution that requires no user involvement or exploring different paths to understand their trade-offs.

We construct and evaluate the application of MCTS and RL on two different techniques (TreeLine and PerfRL). The core tool for instrumentation and testing as used in the state-of-the-art fuzzing techniques allows us to evaluate our contributions effectively. Results on a mix of real-world applications show that our implementation of TreeLine is up to several times faster than a mutational fuzzer at finding expensive inputs for applications with richly structured input (e.g., graph layout) and only modestly slower on applications with unstructured textual input (e.g., word frequency). In general, we can discover bounded-length inputs that trigger exceptionally slow processing in the target applications within few minutes.

This dissertation includes unpublished co-authored material.

CURRICULUM VITAE

NAME OF AUTHOR: Ziyad Alsaeed

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR, USA
Seattle University, Seattle, WA, USA
Qassim University, Qassim, Saudi Arabia

DEGREES AWARDED:

Doctor of Philosophy, Computer and Information Science, 2021, University of Oregon
Master of Science, Software Engineering, 2014, Seattle University
Bachelor of Science, Computer Science, 2008, Qassim University

AREAS OF SPECIAL INTEREST:

Software Engineering
Software Testing
Machine Learning Test Applications

PROFESSIONAL EXPERIENCE:

Graduate Teaching Assistant, University of Oregon, 2021
Graduate Administrative Assistant, University of Oregon, 2018-2021
Lecturer, Qassim University, 2011-2012
Senior Programmer, Security Forces Hospital, 2010-2011
Application Support Officer, Bank Albilad, 2008-2010

GRANTS, AWARDS AND HONORS:

Qassim University Graduate Scholarship, Qassim University, 2012

PUBLICATIONS:

Alsaeed, Z., & Young, M., TreeLine: Finding Slow Inputs Faster with Monte-Carlo Tree Search. 11 pages. (under review for ICSE 2022)

Alsaeed, Z., & Young, M. (2018). Extending existing inference tools to mine dynamic APIs. *In Proceedings of the 2nd International Workshop on API Usage and Evolution (WAPI '18).*, Association for Computing Machinery, New York, NY, USA, 23–26. DOI:https://doi.org/10.1145/3194793.3194797

ACKNOWLEDGEMENTS

First and foremost, I want to thank my research advisor, collaborator, and mentor, Prof. Michal Young, for his guidance in every step through my Ph.D endeavour. Prof. Young is an outstanding and patient mentor who tirelessly helped me develop the necessary research skills to identify a research problem, review papers, and write effectively. His guidance is forever influential to me.

I also would like to thank my committee members Prof. Stephen Fickas, Prof. Christopher Wilson, and Prof. Julie Hessler, for their time and insightful feedback. Their feedback has tangibly improved this dissertation.

Moreover, I gratefully acknowledge the financial support provided by Qassim University. I'm grateful for all the professors who taught me during my undergraduate years and the opportunity they provided me afterward to continue my education.

To my mother Mouthy, my father Hamad, my brother Tariq and my sister Amnah for their everlasting love and support. Without them this would have still been a dream.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

Whether a developer is conscious of the performance of their software as they construct it or not, acceptable software performance is an application requirement. As applications grow in size and complexity, identifying performance issues[1] becomes increasingly challenging. Despite the decades of research within the domain of performance analysis, finding test cases that reveal unknown performance edge cases or confirm the developer's understanding is an active and open research domain.

The application workload (inputs) is an essential element of testing applications' performance. They preserve the whole application context and reflect the actual user experience. However, obtaining these inputs requires deep domain knowledge of the application and hours of testing. The vast research body on input generation for performance testing lacks a good constraint of desired possible worst-case inputs (e.g., search for scaled inputs).

Finding an input pattern of a bounded size that maximizes application execution cost is challenging. Following Lemieux et al. (Lemieux et al., 2018), we define *pathological inputs* as ones that maximize software execution cost subject to a bound on input length. The bound on length implies that the search for scalable inputs is usually trivial and can be explored manually. However, finding an input pattern of a bounded size that maximizes the application execution cost is not.

[1]The phrases performance issue, performance bug, performance problem, and performance improvement opportunity are used interchangeably in the software performance analysis literature. All of these refer to a spot in software in which, if the code block given is fixed; it will improve the application's overall performance. Throughout this document, we will use the phrase "*performance issue*" to refer to such a case.

Recent efforts that use machine learning (Grechanik, Fu, & Xie, 2012) and genetic mutation (Shen, Luo, Poshyvanyk, & Grechanik, 2015) adapt combinatorial test case generation to search for combinations of *values* that trigger bugs. Despite their loose definition of targeted inputs, they require manual identification for search space input properties.

A notable success in finding pathological inputs for a more general class of programs has been shown by mutating arbitrary byte strings for performance testing (Lemieux et al., 2018; Petsios et al., 2017). Fuzzing is a crucial foundation for security testing as attacks often use malformed inputs. However, arbitrary mutations of seed inputs waste significant computational effort and might not be effective beyond testing the applications' parsing module. Finding *algorithmic* performance issues in applications is typically sought in the space of well-formed inputs, which may be obtained more efficiently with generation from a model (e.g., context-free grammar).

Well-known search and learning techniques such as Monte Carlo Tree Search (MCTS) and Reinforcement Learning (RL) can be formalized to use an input generation model to define the search space's scope. Both MCTS and RL have proven their effectiveness in complex search spaces such as board or video games. They can accelerate search toward interesting pathological inputs with an empirically balanced effort between exploration and exploitation of the defined search space.

We adapt both MCTS and RL to search for pathological inputs in two different techniques. Both techniques are capable of finding pathological inputs of tested applications. However, we have much better success in adapting MCTS. Compared to the state-of-the-art fuzzers, we find pathological inputs on more

diverse real-world applications. Specifically, our adaptation of MCTS outperforms fuzzing techniques on applications with well-structured inputs while modest on applications that expect unstructured textual inputs. Furthermore, our technique can scale to a larger desired formation of the same input despite the granular measurement we use for testing (bytes).

The following section elaborates on the problem scope and contributions. We outline the dissertation in Sections 1.2.

1.1 Problem Scope and Contributions

In this dissertation, we focus on generating test cases to expose the performance issues of a wide variety of applications. We use context-free grammar (CFG) as our search scope definer (Section 1.1.1). To constrain the size of the generated inputs to an upper bound following the definition of pathological inputs, we construct an input generation process defining a cost of using CFG tokens and a budget to be used with different search techniques (Section 1.1.2). Moreover, we introduce different methods to scale significantly diverse cost feedback from different applications suitable for search techniques such as MCTS and RL (Section 1.1.3). Finally, we apply some necessary enhancements to RL (Section 1.1.4) and MCTS (Section 1.1.5) to efficiently and effectively drive the search toward expensive workloads.

1.1.1 The Effectiveness of CFG Over Raw Inputs.

The set of sentences generated as derivations of a CFG is far smaller than the sentences generated by randomly mutating a text string. Thus, grammars are a common choice for narrowing the search space explored by testing tools, including grammar-aware and grammar-based fuzzers (Aschermann et al., 2019; Mathis et al., 2019; Srivastava & Payer, 2021).

Nonetheless, a CFG is seldom restrictive enough to narrow the search space to all and only the sentences accepted by an application. For example, if the application is a programming language processor, the set of derivable sentences will include mostly sentences that violate static semantic constraints, such as using only identifiers that have been declared. Some grammar-based testing tools use a CFG augmented with additional constraints for a programming language (*Jsfuzz - Coverage Guided Fuzz Testing for JavaScript*, 2019), some leverage a corpus of valid texts to improve the likelihood that a generated sentence will be accepted (Aschermann et al., 2019; Holler, Herzig, & Zeller, 2012; Mathis et al., 2019). In general, the cost of generating invalid sentences that are useless in testing must be balanced against the cost of slowing the sentence generation process.

In the case of fuzzing tools, in particular, a common operation is *splicing* a portion of one derivation (generally a sentence derived from one non-terminal symbol) into another, creating a new sentence from two or more previously generated sentences or samples from a corpus (Aschermann et al., 2019; Srivastava & Payer, 2021). If we think of searching in the space of sentences derived from the start symbol of a CFG, we can imagine two different kinds of search steps. One search step takes a complete sentence or derivation and "mutates" it to form another (e.g., by splicing). Fuzzing tools generally use this tactic, though not always exclusively. If we conceive of a search tree (typically not represented explicitly), nodes in the search tree are complete sentences or derivations. In a purely generative approach, a search step may be a single derivation step so that nodes in the (conceptual or concrete) search tree are instead phrases. We adopt this finer grain approach for MCTS and RL.

1.1.2 Constraining the Input Size. Bounding the size of test cases dictates that we restrict the derivation step to some input size. Therefore, we associate a notion of cost with grammar elements. Moreover, we associate a notion of budget to the derivation sequence. The cost and budget are formalized on what we call an InputGenerator.

The InputGenerator allows us to use different search methodology to explore inputs of sizes less than or equal to the budget. Moreover, we can use different definitions of cost to define the possible input length. For example, we can use a cost base based on bytes, characters, or even grammar tokens.

1.1.3 Adaptive Feedback. The nature of the problem emphasizes that applications under test can be of different sizes and complexities. Therefore, the cost range for each application is diverse. Moreover, search techniques expect well-defined feedback to drive the search toward the desired goal. For example, MCTS traditionally expect binary rewards to distinguish good from bad paths.

To this extent, we devise different reward functions for different search techniques to drive and adapt to the search progress. Consequently, our techniques can adapt dynamically to applications of different sizes and complexities.

1.1.4 Reinforcement Learning Applicability. Reinforcement learning provides a powerful platform for learning to accelerate in complex environments. Although training in reinforcement learning (or any other machine learning approach) is slow relative to other techniques, it is a promising approach as the overhead of initial training could be carried to subsequent search sessions. Thus, an accurate reinforcement-learning-based test cases generator could be helpful in settings where a continuous integration process is utilized. After initial training sessions, new input generation requires no new heuristics. And as the

application grows, ideally training sessions will require shorter time through transferring knowledge from preceding trained models.

The test case generation problem poses a unique challenge for the reinforcement learning model in that it provides partially observable states. We put together different neural network models to solve the absence of complete derivation context. Moreover, we apply the technique to small target applications such as a word-frequency counter and quicksort algorithm. The model is capable of finding the known worst test cases in a much shorter time than anticipated. However, we identify some scalability limitations before we can apply the technique to larger target applications.

1.1.5 Adaptation of Monte Carlo Tree Search. Monte Carlo Tree Search (MCTS) presents a good model that can find worst-case test cases in a short time. As the size of the search problems becomes larger, MCTS provides an effective strategy to asymmetrically explore the problem space, emphasizing potentially good decision sequences.

With substantial modifications to the original algorithm, TreeLine (an approach based on MCTS) can find complex test cases for various target applications. Evaluating TreeLine on applications that expect well-formed input and others with minimal to no structures expectations demonstrates that TreeLine can find 7.46x expensive inputs compared to the state-of-the-art input-generation fuzzer within the same time constraint. TreeLine is also capable of finding a worst-case pattern for applications with no inputs structure expectations. Moreover, TreeLine will consistently achieve its goal with significantly fewer target-application run cycles.

1.2 Dissertation Outline

The dissertation is outlined as follows. In Chapter II, we provide an overview of related work in the performance analysis domain. We first look into profilers in general to understand the basic methods of performance measurement and the trade-offs in their design. In the same chapter, we cover passive and active input-based profilers to distinguish closely related work. Chapter III provides the fundamental background of context-free grammar, Monte Carlo tree search, and reinforcement learning. We present our work in TreeLine and the basics of input size budgeting and rewarding in Chapter IV. Chapter V presents the work on reinforcement learning and discusses the limitations based on conducted evaluations. We discuss and conclude with the high-level contributions based on the research domain in Chapter VI.

The content in chapter IV is a result of collaboration with co-author (Michal Young) and is not published yet. Ziyad Alsaeed is the primary author of this work and responsible for conducting all the presented analyses.

CHAPTER II

RELATED WORK

In this chapter, we review an extensive body of work in the domain of performance analysis. We aim to classify dynamic performance analysis efforts based on their goals and limitations to identify the research opportunities.
We first detail the general state of software performance analysis (generally known as profiling) in Section 2.1. The established techniques in profiling help define outstanding technical efforts and some trade-offs in performance analysis. Moreover, the reviewed efforts in this part present a good sample of techniques lacking the influence of good test cases. Input-based performance analysis techniques are categorized in Section 2.2. We review the effort made in recognizing the effect of workload on performance analysis. We generally group these into passive input-based analyzers and active input-based analyzers.

We organize each part of the related work chapter as follows:

1. **High Level Look at the State of Software Profiling**: In this section, we review the early established techniques in software profiling such as gprof (Graham, Kessler, & Mckusick, 1982) (Section 2.1.1). The early efforts made in software profiling establish a necessary background of basic technical details on measuring performance and avoiding significant overhead. Next, we review a body of work that focuses on loops as the sole cause of performance issues (Section 2.1.2) and distinguish how these mostly look for scalability issues of software performance. Model-based techniques that dynamically define the oracles for performance testing are reviewed in Section 2.1.3. We show how these model-based techniques work and are usually beneficial in deployed applications only due to possible limited testing workloads. The

work in mitigating the understandability issue of profilers' results is reviewed in Section 2.1.4. Mostly the mitigation of the understandability leads to a trade-off in the technique thoroughness. In the last section (Section 2.1.5), we review some selected domain-oriented papers to show that the essence of profiling limitations are mostly the same despite the differences in targeted domains.

2. **Input Based Performance Analysis**: In this section, we focus on software analysis techniques that recognize the influence of test cases on the thoroughness of profilers. The first group of efforts we categorize (Section 2.2.1) recognizes test cases' importance but does not generate any new inputs (passive). Generally, these techniques offer insight into how the input influences the application performance. For example, define a cost function based on the given input (B. Chen, Liu, & Le, 2016; Coppa, Demetrescu, & Finocchi, 2012; Goldsmith, Aiken, & Wilkerson, 2007; Zaparanuks & Hauswirth, 2012), consider inputs as the high-level configuration to understand their permutations (Siegmund et al., 2012; S. Zhang & Ernst, 2014), or naively provide a stress test of user-defined ranges of inputs to mainly test scalability (Ayala-Rivera, Kaczmarski, Murphy, Darisa, & Portillo-Dominguez, 2018; Küstner, Weidendorfer, & Weinzierl, 2010). Techniques that generate new inputs for performance testing are reviewed in Section 2.2.2. We categorize these based on their core method of input search. Very few techniques use machine learning to generate new inputs (Ahmad, Ashraf, Truscan, & Porres, 2019; Grechanik et al., 2012)—other use genetic algorithms to search for new expensive inputs (Shen et al., 2015). The established machine learning and genetic algorithms lack

the constraint of the pathological input definition, among other limitations. The state-of-the-art techniques use mutational fuzzing to randomly generate new expensive inputs taking advantage of the total increase in test coverage (Lemieux et al., 2018; Petsios et al., 2017). However, as we demonstrate in the related work and other chapters, they suffer from a significant waste in computational effort.

The reminder of the chapter cover each category in details. We conclude this chapter by summarizing the studies and defining research opportunities (Section 2.3).

2.1 High Level Look at the State of Software Profiling

Performance analysis techniques have been developed with different goals in mind. Here, we go over the diverse approaches of software performance analysis, identify their goals and strengths and examine how they address the needs of software engineers. The general established technique will help distinguish basic techniques for profiling (e.g., collecting heuristics) and highlight possible trade-offs in the profilers' design decisions.

Naturally, developers think about performance in terms of how much time a method is taking. In the simplest form of applications, such intuition is valid. However, real-world programs are ever more complex. Applications are constructed of multiple modules, objects, and methods, each interacting with the other. Such complications require more sophisticated software performance analysis techniques that drive the analysis to valuable results and present them in a meaningful way to the developers.

2.1.1 Control-Flow Based Profiling.

Initial distinguished efforts to software performance analysis started as early as the 1980s (Ball & Larus, 1996;

Bentley, 1982; Graham et al., 1982). Graham et al. (Graham et al., 1982) were looking at the software performance analysis in its simplest form. The simplest insight developers are looking for is understanding the method execution time and calls counts at different software architecture abstractions. Provided the software control-flow graph, Graham et al. (Graham et al., 1982) collected the needed information during the application runtime.

Programs are usually composed of multiple parts that different developers write. Moreover, programs usually use external libraries to implement frequently used methods (e.g., data structure libraries). Such composition makes programs intricate and challenging to understand. Graham et al. (Graham et al., 1982) understood such complications and sought to provide results that show the performance cost of routines within the executable program at different abstractions. Such composition can be easier to understand and increase the probability that developers would find appropriate refactoring opportunities for performance gain.

In their solution, Graham et al. (Graham et al., 1982; Graham, Kessler, & McKusick, 2004) explain that merging the two basic measurements of method usage count and time will highlight more significant bottlenecks. Counts are taken within a context (call site), provide the chance to understand the task a method is serving along with its cost. Call site is the identification of a method based on its location within invocation. For example, as shown in Figure 1, if method `foo()` is invoked twice through the program execution, once from within method `bar()` and once from within method `baz()`, then we have two different call sites of `foo()`. On the other hand, the timing profile provides an insight to assess if a method's time consumption is justified given the task it serves despite the number of times it

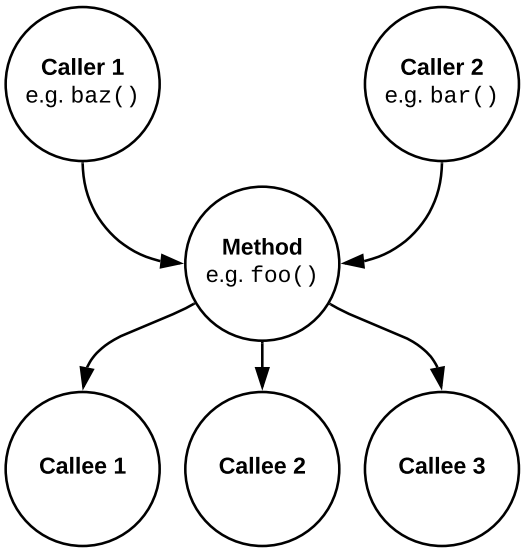


Figure 1. A method within the context of callers and callees.

was called. Once all counts and durations of methods are collected and propagated, they present results in two forms. A flat representation ranking methods based on which contributed the most to the program execution time. Second, provide the ability to examine methods in a *sub-call graph* that shows all the given method callers and callees (see Figure 1). Such representation allows the users to see how the method contributes to its callers' time by knowing how much time a given caller called it compared to all other call instances. Also, it shows which of the callees contributed to its execution time the most by examining how many times a callee was called from that method, given all the calls to that particular callee.

Gprof (Graham et al., 1982) covers the essential cost developers usually think about when analyzing program performance. Moreover, it provides different levels of abstractions for the developer to examine the application. The essential limitation of *gprof* is that it is highly dependent on the given developer test cases to drive the profiling analysis. Developers are not usually performance testing

experts. Moreover, given that unit tests for applications are usually written to assert functional requirement soundness, it is doubtful that the given tests will be helpful for performance testing. Hence, *gprof* would highlight bottlenecks in the programs that are trivial and miss potential threats to program performance. A question comes to mind is would more detailed profiling help mitigate such limitation?

Ball and Larus (Ball & Larus, 1996) focused on the efficiency of the profiling tool itself for fine-grained performance analysis. Path profiling, where a profiler measures how many times a path is executed within a method (Ammons, Ball, & Larus, 1997; Ball & Larus, 1994; Ball, Mataga, & Sagiv, 1998; D'Elia & Demetrescu, 2013; Duesterwald & Bala, 2000; Larus, 1999; Mudduluru & Ramanathan, 2016), is much more precise than block or edge profiling. Moreover, it provides much more detailed information about the method's internal cost compared to *gprof* (Graham et al., 1982) by breaking a method into paths instead of possible method calls from the profiled method. However, a detailed look into a method introduces a significantly higher overhead. It could even be sometimes infeasible for some significantly large applications.

Ball and Larus (Ball & Larus, 1996) introduced an algorithm to enhance the overhead issue that assigns unique IDs to each path to keep a counter of how many times the path gets executed. More precisely, they first convert the Control-Flow Graph (CFG) of methods to Directed Acyclic Graph (DAG). The transformation of CFG creates dummy paths from graph entry to a loop head and from the loop end to the graph exit for each existing loop (back-edges). Loop transformation would reduce the size of the graph and misrepresent loops, but it is necessary for instrumentation. Given the DAG, each edge is assigned an integer value such that

the sum of any path values in the DAG is unique. Such an assignment allows for the unique identification of each path by calculating its ID instead of storing it in memory. Each time a path is taken, the algorithm calculates the path's ID and increment its counter. Hence, it is possible to collect detailed information on large applications.

Duesterwald et al. (Duesterwald & Bala, 2000) focus even more on profiling efficiency. They argue that there is an even higher demand for lower overhead in some profiling cases (e.g., just-in-time compilation). A solution is to impose less profiling to gain more knowledge within a smaller space and time. The key idea is to identify a threshold to determine a path head is hot. Once a path head is identified as hot, no more profiling is made, and a prediction is made about its tail using a dynamic optimization system. The argument is that shorter intervals of path evaluations help reduce the overhead while maintaining similar hot path predictions.

The work established by Ball and Larus (Ball & Larus, 1996) determines bottlenecks in applications in terms of execution complexity. However, it does not take into account the application usage of memory. Mudduluru et al. (Mudduluru & Ramanathan, 2016) consider such a problem and established a control-flow profile (called object-flow profile) to track object (data) creation and access based on Ball and Larus numbering. For each allocation, Mudduluru et al. (Mudduluru & Ramanathan, 2016) will preserve a control-flow graph from the allocation site to the locations where the object got used (maintaining a count for each edge). The intuition is that hot paths in such flow profiles will help locate inefficiently used memory spaces.

Efficient and highly detailed profiling techniques (Ball & Larus, 1996; Duesterwald & Bala, 2000; Mudduluru & Ramanathan, 2016) made it possible to collect count information of path profiles regardless of the application size or complexity. However, a more detailed view makes it evident that designated performance test cases are necessary compared to *gprof* (Graham et al., 1982) as more precise paths need to be exercised. The efficiency and preciseness of performance analysis tools do not ensure the profiling results' fruitfulness or understandability. If different, it can help highlight where additional testing might be necessary.

2.1.2 Loop Focused Profiling. In addition to the need to overcome limitations in interpreting loops based on control-flow profiling (D'Elia & Demetrescu, 2013), the common knowledge in the field and few study papers (Jin, Song, Shi, Scherpelz, & Lu, 2012; Nistor, Jiang, & Tan, 2013) asserts that most complex and hard to fixed performance issues happened within special forms of loops. Loops can be in different forms. For example, loops can be in the simple form of language provided keywords such as `for` or `while` loops, as a recursive method or even less clear as mutually recursive methods (see Figure 2). Studies (Jin et al., 2012; Nistor, Jiang, & Tan, 2013) assert that performance issues are even more severe within nested loops. Such a conclusion prompted different research efforts to focus the program analysis on loops where most performance issues occur.

Focusing on loops when analyzing program performance introduces a couple of advantages. First, it reduces the number of instrumented instructions to those within known loop patterns. Thus, reducing the overhead. Second, it guides the studies of program performance toward the actual symptom of performance issues.

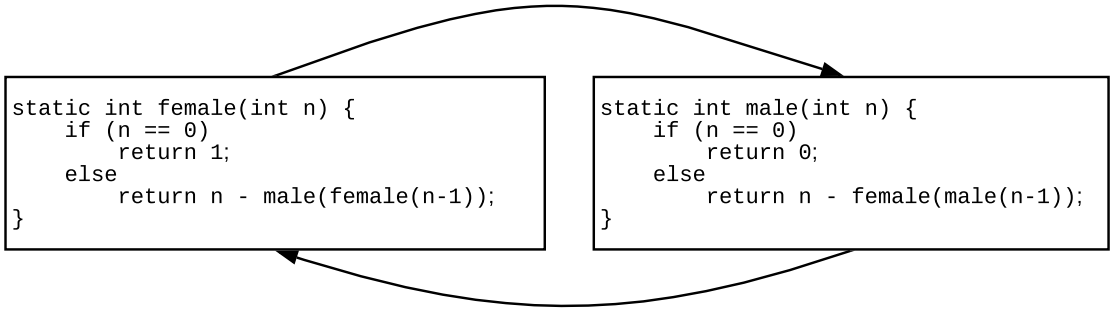


Figure 2. Java example of mutually recursive methods forming a loop based on Hofstadter female and male sequences.

In this section, we go over distinguished efforts (Dhok & Ramanathan, 2016; Nistor, Song, et al., 2013; Song & Lu, 2017; Xiao, Han, Zhang, & Xie, 2013) that explicitly study the effect of unique loop cases on performance.

```
1  // Simplified from the XYPlot class in JFreeChart
2  public void render(...) {
3    for (int item = 0; item < itemCount; item++) { // Outer Loop
4      renderer.drawItem(...item...); // Calls drawVerticalItem
5    }
6  }
7  // Simplified from the CandlestickRenderer class in JFreeChart
8  public void drawVerticalItem(...) {
9    int maxVolume = 1;
10   for (int i = 0; i < maxCount; i++) { // Inner Loop
11     int thisVolume = highLowData.getVolumeValue(series, i).intValue();
12     if (thisVolume > maxVolume) {
13       maxVolume = thisVolume;
14     }
15   }
16   ... = maxVolume;
17 }
```

Figure 3. Computation redundancy performance issue found on `JFreeChart` as shown in (Nistor, Song, et al., 2013).

Nistor et al. (Nistor, Song, et al., 2013) established the general idea of monitoring instructions behavior within loops. In particular, they look for nested loops that do redundant work. For instance, the code shown in Figure 3 illustrates

a severe computation redundancy that is hard to find (Nistor, Song, et al., 2013). As the outer loop iterate over all items in a data set (line 3), it calls the method `drawItem` which in turn calls the method `drawVerticalItem`. The inner loop within `drawVerticalItem` (line 10) also iterates over all items in the data set to find the one with maximum volume. As the volume in the data set does not change over the different loops, such computation is redundant and found to be causing the rendering to freeze. The performance issue is fixed by caching the maximum volume value within the outer loop to avoid redundant work.

Nistor et al. (Nistor, Song, et al., 2013) monitored memory access within the identified nested loops to automatically find redundant computations. If a group of instructions accesses similar memory values across iterations, those instructions probably compute similar results. Thus, there is a performance issue. Nistor et al. (Nistor, Song, et al., 2013) introduced a tool called Toddler that implements loop monitoring. Toddler first statically analyzes the code searching for loops and assigning unique IDs to each loop. Then, Toddler instruments the code by inserting triggers to identify loops at three major stages: loop starts, loop iteration starts, and loop end. Toddler identifies a read instruction by both the static occurrence of the instruction in the code and the dynamic context (call stack) in which the instruction is executed and calls it IPCS (Instruction Pointer + Call Stack). A sequence of IPCS is collected for each loop within a nested loop structure (outer or inner). IPCS sequences are eventually compared given a threshold looking for *read* values similarities across loop iterations. If there is any such IPCS sequence, Toddler reports a performance issue.

Slightly different from Toddler, Song and Lu (Song & Lu, 2017), tackled the problem based on prior knowledge of the performance issues symptoms within

loops. Initially, Song and Lu (Song & Lu, 2017) studied known performance issues within loops to provide a taxonomy of the root causes of the inefficiencies. Their study resulted in two notable classifications of loop inefficiency resultless loops and redundant loops. Resultless loops do many computations but do not show any side effects. Redundant loops do repetitive computations (same inputs and outputs on some of the iterations). Song and Lu (Song & Lu, 2017) argue that using the taxonomy to look for suspicious loops helps focus the search on a smaller set of loops. To validate their hypothesis, they developed a tool called LDoctor that uses static analysis techniques to identify potential loops. It also uses dynamic analysis techniques along with sampling to analyze applications under a given workload. The tool proves to be efficient and accurate, but only for the given limited search scope.

Toddler (Nistor, Song, et al., 2013) and LDoctor (Song & Lu, 2017) represent a focused look at program performance analysis. However, they do not explore instrumented application beyond the developer-provided test cases. Thus, similar to old basic techniques (Ball & Larus, 1996; Duesterwald & Bala, 2000; Graham et al., 1982; Mudduluru & Ramanathan, 2016), they inherit the limitation of the developer's testing assumptions during analysis. As a mitigation, the same idea of exploring loops has been explored with the attempt to stress loops given new inputs.

In order to overcome the limitation of finding new and unanticipated performance issues within loops, Xiao et al. (Xiao et al., 2013) analyzed application given a set of test cases. The test cases provided are assumed to be expressing the application functionality. For example, for a compression algorithm, a test case or a scenario, as Xiao et al. (Xiao et al., 2013) calls it, can be a task a user can

complete and a set of parameters to manipulate. This restriction makes it easier to create new inputs by manipulating the parameters and recording the ones that affect the performance. However, even with such restrictions, the approach is limited in finding scalable inputs only. Scalable inputs are best described as ones that increase the test case's size but do not manipulate the input structure. For example, when compressing files, they are only capable of increasing the number of files to compress, not the nature of the files. Thus, they only evaluate one aspect of input manipulation.

Dhok and Ramanathan (Dhok & Ramanathan, 2016) presented another technique that identifies the main limitation of Toddler (Nistor, Song, et al., 2013) and others. They attempt to generate more tests based on the seed tests provided by developers or existing random test generators (Pacheco & Ernst, 2007). First, they generate methods summaries based on the given tests. The methods summaries are composed of information about the presence of a loop, the objects traversed in each loop, and the methods invoked within the loop. Second, they identify methods with potential nested loops. Method detection is based on looking at the call graph for symptoms of known patterns (Wert, Happe, & Happe, 2013) that lead to bad performance using similar techniques presented in (Song & Lu, 2017). Finally, they generate performance-focused tests for the given methods with emphasis on the scale of the inputs. This approach overcomes the manual parameter identification presented by Xiao et al. (Xiao et al., 2013). However, in addition to finding scalable inputs only, the introduced approach's main limitation is in its assumption that initial given tests will lead to interesting performance issues. Moreover, it is limited to known symptoms of lousy performance within

the code. Thus, it can reveal some performance issues, but at the same time, skip others.

The loops focused technique provides a more valuable understanding of the performance issues in generals. However, they either are not exploring unanticipated issues or mainly provide scalable inputs to a small set of known performance issue patterns (Wert et al., 2013). These efforts (Dhok & Ramanathan, 2016; Nistor, Song, et al., 2013; Song & Lu, 2017; Xiao et al., 2013) and basic ones (Ball & Larus, 1996; Duesterwald & Bala, 2000; Graham et al., 1982; Mudduluru & Ramanathan, 2016) are limited in asserting developers' understanding of the program by profiling applications based on developer's test cases. If an unanticipated performance issue exists, neither passive nor active presented performance analysis tools will help capture them.

2.1.3 Model Based Profiling. A unique characteristic of non-functional requirements (e.g., performance) is the difficulty of setting a goal for the requirement. Such nature made it necessary to explore methods that would assist in establishing some boundaries that define the system performance goals. For example, in a word processing application, it is expected for a letter to appear instantly on the screen as soon as the user hits the letter key. However, in addition to the difficulty of thinking about every possible scenario in the application in terms of performance, it is hard to put a number on such cases. Moreover, if a task becomes the focus of the developers, it is usually easier to understand if the current performance is acceptable given some prior knowledge than writing specifications that define the expected performance. Performance modeling is a formal way to mitigate the issue and establish rules about program expected performance (oracles) (Balsamo, Di Marco, Inverardi, & Simeoni, 2004). Performance models

define the precise performance boundaries of an application or its modules. Given the advantages modeling provides in understanding the performance, many (Balsamo et al., 2004; Brünink & Rosenblum, 2016; Hoefler, Gropp, Kramer, & Snir, 2011; Koziolek, 2010) attempted to find such models automatically.

Brünink and Rosenblum (Brünink & Rosenblum, 2016) made a notable effort in this area. They automate the finding of performance models based on actual runs of the program then summarize these models to maintain performance assertions about the program. The intuition is that such assertions would help monitor the application's performance and get triggered upon any performance deviations. To find these performance models, Brünink and Rosenblum (Brünink & Rosenblum, 2016) monitor an application during a given run (usually a deployed application) to obtain runtime insight of given methods (usually hot methods). Then they analyze the collected runtime data to check if it fits different runtime clusters and stable (i.e., no new unclassifiable data is further showing up). If such data for a given method exists, they collect call stack information to relate them to these different runtime clusters. The process then repeats for the given method callers until no other interesting methods are introduced.

The generated performance models are usually large and hard to understand. Therefore, Brünink and Rosenblum (Brünink & Rosenblum, 2016) introduced the idea of finding performance assertions. These performance assertions are the shortest possible descriptive paths in the form of an expression to the given method, and the time it took relative to the path. The resulting set of assertions is maintained for future tests or actual use monitoring. Any execution that breaks a given assertion is a performance issue.

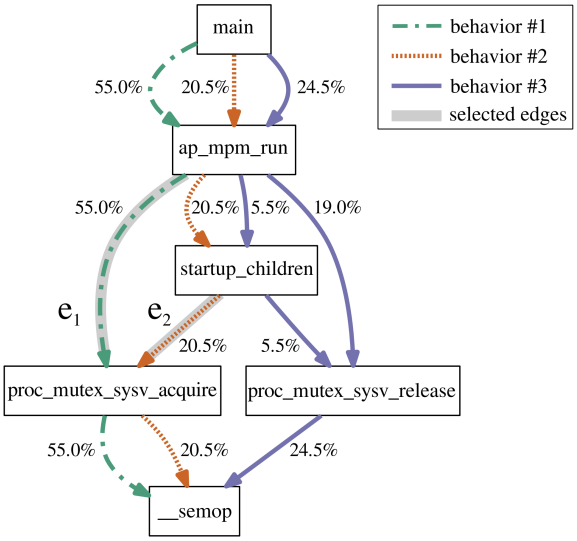


Figure 4. Performance model of calls to method `__semop` in Apache 2.0.64 (Brünink & Rosenblum, 2016).

An example of the generated model given Brünink and Rosenblum (Brünink & Rosenblum, 2016) approach is illustrated in Figure 4. The model shows the different summarized context to the method `__semop` in Apache v2.0.64. This automatically learned model state that calls which does not involve edge e_1 or e_2 take less than 70ms. Otherwise, executions would take less than 70ms in 41.9% of the cases if it includes edge e_1 and in 35.5% of the cases if it includes edge e_2. Hence, assertion about the method `__semop` for newer versions can be as simple as if $e_1 \wedge \neg e_2$ then $t < 70ms$ in 41.9% of the cases, if $\neg e_1 \wedge e_2$ then $t < 70ms$ in 35.5% of the cases and if $\neg e_1 \wedge \neg e_2$ then $t < 70ms$ in 99% of the cases where t is the time spent to call the method `__semop`. Using expressions to capture the system behavior given new changes to the code is precise and meaningful feedback to developers.

Brünink and Rosenblum (Brünink & Rosenblum, 2016) approach provides a method that would generate test oracles. However, obtaining valuable test oracles occurs only if the tool is used with deployed applications. Valuable workloads are only present when actual users are using the system. Therefore, actual performance

insights appear only if the application is in use. And as stated before, at this stage of the application life cycle, performance issues are usually unaffordable.

Hoefler et al. (Hoefler et al., 2011) presented a method to build performance models for parallel applications. Nevertheless, the essence of their work is also applicable to non-parallel applications. Hoefler et al. (Hoefler et al., 2011) offer to introduce performance-modeling techniques in every software development stage (e.g., design, implementation, and testing). However, such an approach is unrealistic for a more agile[1] project management methodology as design efforts are minimal. The guidelines they introduced from their study apply to user-based applications. They obtained the guidelines from experimenting with performance modeling on a set of subject applications. These guidelines can differ based on the point of view when looking at an application. For example, identifying input parameters that influence the runtime considers application workload, but determining communication patterns considers application structure. Although no automation was proposed to generate performance models, they established a foundational systemic approach of performance modeling for others to use (e.g., the work developed by Brünink and Rosenblum (Brünink & Rosenblum, 2016)).

In general, performance modeling approaches (Balsamo et al., 2004; Brünink & Rosenblum, 2016; Hoefler et al., 2011; Koziolek, 2010) are essential in identifying boundaries that would help test the applications' performance. However, those boundaries can be vague or difficult to understand. Hence, lower chances of adaptation. Moreover, identifying those boundaries is difficult. Automating such tasks requires thorough unit tests and workloads that would lead to potential performance issues within a given application.

[1]Agile is a widely adapted software development approach under which requirements and artifacts grow and change together.

```
(? i :( j |(&# x ?0*((74) |(4 A ) |(106) |(6 A ) ) ;?) )([\ t ]|(&((# x ?0*(9|(13) |(10) | A | D )
 ;?) | ( tab ;) |( newline ;) ) ) ) *( a |(&# x ?0*((65) | (41) |(97) |(61) ) ;?) ) ([\ t ]|(&((#
x ?0*(9|(13) |(10) | A | D ) ;?) |( tab ;) |( newline ;) )) ) *( v |(&# x ?0*((86) |(56) |(118)
|(76) ) ;?)) ([\ t ]|(&((# x ?0*(9|(13) |(10) | A | D ) ;?) | ( tab ;) |( newline ;) ) ) ) *( a
|(&# x ?0*((65) | (41) |(97) |(61) ) ;?) ) ([\ t ]|(&((# x ?0*(9|(13) |(10) | A | D ) ;?) |( tab
;) |( newline ;) ) ) ) * ( s |(&# x ?0*((83) |(53) |(115) |(73) ) ;?) ) ([\ t ]|(&((# x ?0*(9|(13)
 |(10) | A | D ) ;?) | ( tab ;) |( newline ;) ) ) ) *( c |(&# x ?0*((67) | (43) |(99) |(63) ) ;?)
) ([\ t ]|(&((# x ?0*(9|(13) |(10) | A | D ) ;?) |( tab ;) |( newline ;) ) ) ) * ( r |(&# x
?0*((82) |(52) |(114) |(72) ) ;?) )([\ t ]|(&((# x ?0*(9|(13) |(10) | A | D ) ;?) |( tab ;) |(
newline ;) ) ) ) *( i |(&# x ?0*((73) |(49) |(105) |(69) ) ;?) ) ([\ t ]|(&((# x ?0*(9|(13) |(10)
| A | D ) ;?) |( tab ;) |( newline ;) ) ) ) *( p |(&# x ?0*((80) |(50) |(112) |(70) ) ;?) )([\ t
]|(&((# x ?0*(9|(13) |(10) | A | D ) ;?) |( tab ;) |( newline ;) ) ) ) *( t |(&# x ?0*((84) |(54)
|(116) |(74) ) ;?) ) ([\ t ]|(&((# x ?0*(9|(13) |(10) | A | D ) ;?) |( tab ;) |( newline ;) ) ) )
*(:|(&((# x ?0*((58) |(3 A ) ) ;?) |( colon ;)) ) ) .)
```

Listing 2.1 Input generated by SlowFuzz (Petsios et al., 2017) to demonstrate a special input that causes a slowdown in PCRE (pcre.org, 2019) regular expression matching library. Although the input demonstrates a performance issue, it is not easy for a developer to understand, and therefore not very helpful in addressing the issue.

2.1.4 Actionable Profilers.

An essential strength in any given profiling technique is its ability to simplify its results to facilitate its comprehensibility by developers. Simplifying the results is not an easy task since it requires predicting the developer's needs. Moreover, profilers have to provide different abstraction levels at which the developer can observe the results. For example, Listing 2.1 shows a special input that exhibits a 20% slowdown in the PCRE (pcre.org, 2019) regular expression matching library. From the example, it is clear how it is hard to link the input to the root cause of the performance issue. Lack of improving the result understandability could lead to losses in performance optimization opportunities. Established efforts (Della Toffola, Pradel, & Gross, 2015; Nistor, Chang, Radoi, & Lu, 2015; Selakovic, Glaser, & Pradel, 2017) tried to analyze applications with the goal of understandability in mind.

A simple method to measure the understandability of the results is to make them actionable. Actionable results are automatic syntactically valid suggestions of performance issues fixes a developer can approve or reject. Based on the developer's longtime understanding of the code, they can approve the fix without even understanding the performance issue if they agrees with the changes' functional

integrity. Such recommendations take out the barrier of explaining the performance issue by providing a code change that guarantees performance improvement.

Nistor et al. (Nistor et al., 2015), are the first to ask the question of *how likely are developers to fix a detected performance issue*? They study the question in relation to many attributes such as how likely is fixing a performance bug would introduce a functional bug or break a good coding practice. Most importantly, they study the behavior of the developers given the effort and time needed to understand the performance issue as well as understanding the trade-off on other modules of the software if any. They found that it is less likely to fix performance issues if it is hard to understand the issues or relate them to other modules in the software. Given this understanding, they settle on statically finding bugs that have non-intrusive fixes. Primarily, they focus on loops that waste computation after a certain condition is satisfied. The fix for such performance issues is simple and a developer needs only to check if a condition is satisfied to break off the loop. Nistor et al. (Nistor et al., 2015) showed that their reported issues have a high adaptation rate given that they are easy to understand. However, even if the found performance issue would introduce a significant speedup in the software, the search scope is very limited. Thus, it is clear that there is a trade-off between the performance analysis tool comprehensibility and the result understandability.

An important question comes to mind when provided with a solution that makes change suggestions is why would not the profilers make the changes without even going back to the developer? As the profiler ensures performance enhancement, they should make the change. To the contrary of compiler optimization, profilers cannot ensure preserving the program integrity if a change is applied. The suggested changes are always syntactical sound but not necessarily

semantically. Preserving semantic safety means the performance analysis technique would be very limited. Performance analysis techniques targets changes that are beyond the compiler ability to optimize. Therefore, human judgment is necessary to approve the changes.

Selakovic et al. (Selakovic et al., 2017) present a notable effort in providing actionable results. The focus of the technique is on finding the optimal order of logical expressions. Given a logical expression (e.g., `if` or `switch` statements), what would be the optimal order of the expressions to evaluate and reach a decision. For example, if we have the statement `if (a > 0 && b == 1)`, based on the program executions which expression "`a > 0`" or "`b == 1`" would be `false` most of the times. Whatever expression that usually yields `false` more frequently should be evaluated first. The intuition is that if a low-cost expression evaluates mostly to `false`, you would want to check it first to avoid wasting calculations on other expressions. The example given is trivial, and changing its order would not affect the performance. However, other logical expressions could be large and have a tangible opportunity. In addition, the authors (Selakovic et al., 2017) are looking for the cumulative gain from these small changes.

Selakovic et al. (Selakovic et al., 2017) will run all available expressions after preserving the program state. Given the available test cases, the profiler will collect data about the expressions' cost and results for each traced execution. The cost is defined as the number of executed branching points within each expression rather than time. Measuring the time for such expression is challenging as these are usually fast operations. Given the expressions' common results (`True` or `False`) and the execution cost, the profiler will compute the computational cost of all possible orders of the expressions. Once they chose a given order and proved to be making

the program faster by executing it, the author will suggest the new order to the developer as an actionable suggestion.

The work introduced by Selakovic et al. (Selakovic et al., 2017) achieves a high degree of result simplification and understandability. However, the trade-off between the understandability of the results and the value of the fix is significant. The enhancement when applying all the semantics preserving changes based on their evaluations is between 2.5% and 6.5% at the application level. Given the significant size of the evaluated applications (e.g., Apache Struts), this is a meager improvement. Moreover, these changes are highly dependent on the given workload in the unit tests. As we know, a significant issue is that it is hard for developers to anticipate real-world workloads. Thus, any change in the workload at deployment might render the performance changes obsolete.

Della Toffola et al. (Della Toffola et al., 2015) established another distinguishable work that attempts to be actionable. They introduce a technique called *MemoizeIt* to look for memoization opportunities. As code might suffer from redundant computations that lead to program performance issues, these are good opportunities to optimize the code. Traditional profilers might miss or low-rank such opportunities. Locating such redundant computations based on the given inputs and outputs of methods could reveal memoization opportunities that enhance the program's overall performance (Della Toffola et al., 2015; Guo & Engler, 2011; Xu, 2012).

MemoizeIt narrows down the tracked elements into the target object, parameters, and return results. Nevertheless, such a profiling technique can be significantly expensive. Therefore, the authors introduced an iterative approach to monitoring the program. First, run the program with light profiling that

records the execution time of each method. A method that does not have an expensive computation is discarded as it is less likely to be optimized even further. Second, increase the depth of object exploration gradually to look for structure inconsistencies based on a flattened object representation (flattened representation is nothing but representing objects' types and values in nested arrays). For example, if a method is called twice wherein the first call returned an object with two fields but in the second an object with three fields, then the method is dropped from the list for further analysis. Such iterative trimming and depth increasing allows *MemoizeIt* to maintain efficiency while increasing accuracy gradually. It is important to note that as the depth can be unbounded for *MemoizeIt*, the authors observed that object exploration of depth-2 is sufficient to be accurate. Third, *MemoizeIt* computes a cache-hit rate as it iteratively monitors candidate methods. They discard methods that go below a user-defined hit-rate threshold for each iteration. Such computation helps to maintain a list of potential methods that are more likely to benefit from memoization.

In addition to profiling methods based on their input and output, cluster them, and rank the method based on their potential performance gain, *MemoizeIt* suggests a fix to the developer to simplify the result. In order to provide a suggestion on how to fix a method, *MemoizeIt* simulates different ways of fixes while validating the program integrity after applying the fix based on the given unit tests. Memoization fixes usually happen globally or locally (instance level) and are of multi (storing more than one input and output value) or single cached input-output pairs. Combining these possibilities required the authors (Della Toffola et al., 2015) to simulate four different fixes. Given the result with the highest hit rate,

MemoizeIt considers it the best possible change and suggests it to the developer as an actionable fix.

There are other memoization techniques (Infante, 2014; Nguyen & Xu, 2013; Xu, 2012), but taking only *MemoizeIt* (Della Toffola et al., 2015) as a representative technique, we can identify the major limitations in memoization. First, given the conducted tests by Della Toffola et al. (Della Toffola et al., 2015), it is clear that the number of such memoization opportunities is minimal. From eight different applications (DeCapo benchmarks (Blackburn et al., 2006)), they found only nine distinct memoization opportunities. Second, most of the found memoization opportunities are workload-dependent. Thus, regardless of its performance gain, the fix might not be of significance when deployed. We can generalize these limitations to all memoization techniques.

Our discussion about actionable profilers (Della Toffola et al., 2015; Nistor et al., 2015; Selakovic et al., 2017) shows a clear trade-off between the result understandability and performance optimization opportunities. To make results actionable, they must be simple and follow a narrow a well-defined pattern. Moreover, simple fixes are less likely to capture significant or unanticipated performance issues.

2.1.5 Domain Oriented Performance Analysis. Some performance analysis techniques have been targeting specific domains such as mobile devices or supercomputing. Nevertheless, the fundamental challenges are usually the same. In this section, we cover efforts that target different but related techniques to our work. The main domains are related to mobile, parallel computing, or scientific computing.

Given that smartphones are widely popular nowadays, the challenges smartphone application developers face in tuning the performance of their applications are fundamentally similar to known performance analysis challenges (Liu, Xu, & Cheung, 2014). However, the developer's focus in parts of the applications where performance issues might appear is slightly different. For example, many of the performance issues found on smartphone applications are UI-related. Because the UI actions trigger asynchronous executions in many different ways, Kang et al. (Kang, Zhou, Xu, & Lyu, 2015; 2016) introduced a method to track and profile these executions by categorizing them into small sets. Similarly, Brocanelli et al. (Brocanelli & Wang, 2018) focus on the expensive operation happening on threads other than the application's main thread but cause performance issues. These techniques are different in directing the analysis to specific application areas but are fundamentally similar to all other performance analysis techniques in relation to data collection and analysis.

Performance analysis of supercomputers or more generally distributed systems has its different emphasis on what to profile. However, techniques that target supercomputers are also fundamentally similar to techniques that target sequential applications. Frameworks like TAU (Shende & Malony, 2006), in addition to providing a performance analysis of distributed systems, it target providing an architecture-specific insight about the application performance (Boehme et al., 2016; Ofenbeck, Steinmann, Caparros, Spampinato, & Püschel, 2014). Some techniques try to measure the software performance given the supercomputer capabilities (Ofenbeck et al., 2014) and highlight if the systems have been used to their full extent or not. Other (Boehme et al., 2016) tackle collecting and unifying the knowledge about the data in a more modular supercomputer

environment. Some techniques like the one presented by Herodotou and Babu (Herodotou & Babu, 2011) focus on providing an insight to the developer about the system performance under a given workload by manipulating multiple configuration options.

More focused on parallel programs, Curtsinger and Berger (Curtsinger & Berger, 2015) attempt to provide developers with information about the expected performance gain if a particular method is fixed. They show that by slowing other threads whenever the method is executed on one of the tracked threads. The slowdown of threads simulates the performance gain of fixing the targeted part of the code.

A fundamental difference in performance analysis techniques for supercomputers or closely related applications is that the results' understandability is not a significant concern. Users of supercomputer analysis tools are usually more experienced and knowledgeable about their applications. Moreover, the use case for those systems like what Herodotou and Babu (Herodotou & Babu, 2011) presented is usually task-oriented. Meaning, the user has a given location in mind about the software and needs comprehensive insight about its interactions and cost. Thus, there is less to no emphasis on understandability in such techniques.

The work focused on mobile devices or supercomputers is similar to other performance analysis techniques designed for sequential computers. They are distinguishable in that they are domain-specific. Given the knowledge about a given domain and the probability of where performance issues might appear, they tailor the fundamental methods of profiling to exploit such performance issues. However, these established works do not solve some fundamental challenge such as identifying worst-case workloads.

2.2 Input Based Performance Analysis

Inputs (i.e., workload or test cases) are an essential factor in any software analysis tool. The absence of expressive inputs that trigger worst-case complexity within the performance analysis field is an inherent issue. Different sets, sizes, and orders of inputs can express different limitations of code blocks. Moreover, the performance of a code block based on a given input can significantly differ according to the whole program state (i.e., context). For instance, `Mozilla Bug #490742` (Jin et al., 2012) illustrates such a performance issue. The reported (and fixed) performance issue appeared only when users tried to bookmark 20 or more pages at once using the *Bookmark-All* functionality on Mozilla Firefox. Without going into details of how the performance issue was introduced and fixed, it is easy to see how a given case could escape the software testing designed toward testing the method's functionality. Load testing (Burnim, Juvekar, & Sen, 2009; Cadar, Dunbar, & Engler, 2008; P. Zhang, Elbaum, & Dwyer, 2011) could be a probable solution for such simple performance issues. However, the dimensionality of inputs complexity is only assumed to be larger in most cases. There exist performance issues that does not occur because of the size of the inputs. Thus, having a meaningful input that expresses the application performance is an essential and challenging problem.

The number of different possible paths within an application can be significantly large. Static and dynamic analysis tools provide insightful feedback (Tikir & Hollingsworth, 2002) and module classification (Grant, Cordy, & Skillicorn, 2008) about the application under test. For example, code coverage tools (Tikir & Hollingsworth, 2002) can easily report that the path `ABDE` in Figure 5 has never been taken. Such insight allows the developers to write tests that express

the functionality of such a path. However, presented performance analysis tools do not help the user distinguish how tested paths `ACE` and `ABCE` from Figure 5 differ. Because of the given input data by the user, a passive performance analysis tool could mislead the developer into believing that path `ACE` has a performance issue and thereby miss actual performance optimization opportunities.

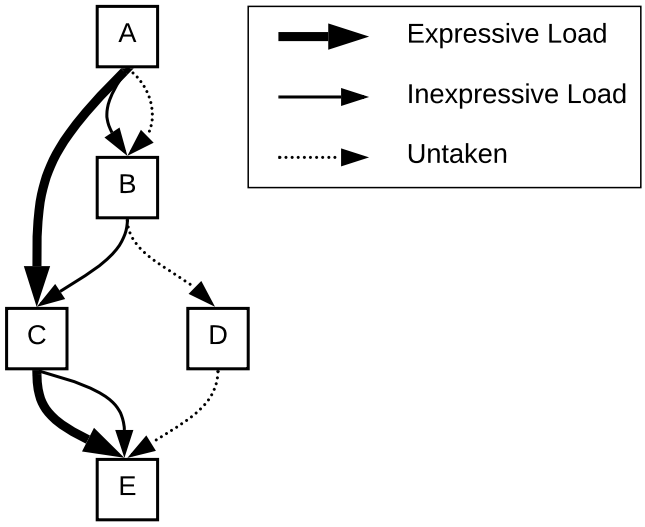


Figure 5. Example of different paths traversed based on testing data.

In this section, we present proposed performance analysis tools that look into the influence of the input data on the performance of applications under test. There are two major types of input-based performance analysis tools. One tries to provide insight on how the inputs influence the application performance based on provided inputs (Luo, 2016b; Zaparanuks & Hauswirth, 2012) (Section 2.2.1). In contrast, the other tries to explore new inputs that would express the application's performance beyond what the developer anticipated (Grechanik et al., 2012; Lemieux et al., 2018; Petsios et al., 2017; Shen et al., 2015) (Section 2.2.2).

2.2.1 Input Influenced Insight. Several performance analysis techniques realize the significant influence of inputs on how applications are

performing (Ayala-Rivera et al., 2018; Coppa et al., 2012; Küstner et al., 2010; Luo, 2016a; 2016b; Shen et al., 2015; Siegmund et al., 2012; Zaparanuks & Hauswirth, 2012; S. Zhang & Ernst, 2014). These approaches differ in how they define the inputs and analyze them. For example, some efforts (B. Chen et al., 2016; Coppa et al., 2012; Zaparanuks & Hauswirth, 2012) try to define the cost functions of the analyzed methods based on the given inputs. Some others (Siegmund et al., 2012; S. Zhang & Ernst, 2014) narrowly define input as the possible configuration a user could change for a given application and provide insight into which combinations maximize the performance. Others (Ayala-Rivera et al., 2018; Küstner et al., 2010) try to find application bottlenecks based on the given inputs and provide the developer with insightful feedback to provide better inputs manually. Despite their differences, all of these techniques are either passive techniques where they do not look for new interesting inputs or rely on the developer to drive the input search.

2.2.1.1 Defining Cost Functions. Algorithm researchers and software engineers for scientific applications usually use cost functions when analyzing algorithms complexity. Cost functions are usually asymptotic lower and upper bound representations of the algorithm time or space cost. Such insight can be helpful to understand what inputs could lead the application under test into an unsatisfying performance. Using cost functions, software engineers overcome the input anticipation issue by confirming whether the inputs they expect fall within a bad performance portion of the cost function or not. Cost functions are not necessarily trivial to obtain. Moreover, they are even more challenging when the input's structure is multifaceted for real-world applications. Hence, performance analysis researchers (B. Chen et al., 2016; Coppa et al., 2012; Goldsmith et al.,

2007; Zaparanuks & Hauswirth, 2012) attempted to find these cost functions automatically based on program runs.

Automating the construction of cost functions is not trivial. One of the primary challenges in automating the construction of cost functions is determining the inputs' basic blocks of an algorithm. Zaparanuks and Hauswirth (Zaparanuks & Hauswirth, 2012) established a technique that identifies all loops in a control-flow graph and recursions in the program's call graph to locate areas where performance issues might occur. Moreover, they use the execution count of loops as the cost instead of measuring time to avoid significant overhead. Zaparanuks and Hauswirth (Zaparanuks & Hauswirth, 2012) essential shortcoming is in determining the algorithm's or method's inputs. Either they limit what they considered inputs to field references to data structures accessed within the method execution or external input files. Even more, their approach has issues with algorithm inputs based on primitive data. Because the cost interpretation of primitive data types can differ (e.g., an integer can be seen as a *number of digits* on the schoolbook multiplication algorithm or a *value* on a factorial algorithm), their approach focuses on one aspect only.

Conceiving primitive inputs is not the only noticeable limitation on Zaparanuks and Hauswirth (Zaparanuks & Hauswirth, 2012) work. Experimental algorithmic techniques such as Zaparanuks and Hauswirth (Zaparanuks & Hauswirth, 2012) take portions of the code and test it extensively on different input sizes. While such approaches can provide valuable insight into the code portion, it suffers from studying those small portions outside of their context (the system as a whole) (Sumner, Zheng, Weeratunge, & Zhang, 2010; Zhuang, Serrano, Cain, & Choi, 2006). Much of the performance issues that escape testing combine

multiple and complex interactions between different portions of the system. Taking a method out of its context or looking at it within a static context does not provide complete insight into the application performance.

Coppa et al. (Coppa et al., 2012) understood the context issues for performance analysis in general and approaches that try to generate cost functions in particular. They mitigated the limitations of existing approaches that automatically measure the performance of the routines as a function of their input size by looking at it within the actual context of the software.

Before looking into the context issue, it is important to look into how Coppa et al. (Coppa et al., 2012) approached the fundamental challenge of defining the inputs to track application performance. They introduced a metric called Read Memory Size (RMS). They define RMS as the number of distinct memory cells first accessed by a method or by a descendant of the method in the call tree, with a read operation. Such fine-grained insight can be obtained using tools such as Valgrind (Nethercote & Seward, 2007). Coppa et al. (Coppa et al., 2012) argue that calls to memory by a function for the first time (never accessed before) with a read operation contain the input values of the function. Conversely, if a cell value is first written and then read by the function, the value is not part of its input as it was determined by the function itself.

Although the RMS definition can limit the number of tracked inputs, we think this is a distinguished contribution to defining inputs in the domain of performance analysis. This approach in defining inputs breakdown complex user-defined objects by representing them in their simplest form. The overhead of the technique is significantly high, but that is an expected trade-off between the details and cost of any software analysis tool.

Having the RMS defined, Coppa et al. (Coppa et al., 2012) define the performance analysis technique as the collection of RMS for each method encountered in the program. For example, for method `foo()`, they find the set $N_{foo} = \{n_1, n_2, ...\}$ of distinct RMS values on which `foo()` has called during the execution of the program. For each estimate of the input size $n_i \in N_{foo()}$, they collect the number of times the method is called under that input, the maximum and minimum cost for the method to be executed based on the observation, the sum of all the costs observed for the given method, and the sum of the costs' square.

This definition of how Coppa et al. (Coppa et al., 2012) collect inputs makes their approach context-sensitive. Because they separated how they collect input information from the method's location, they captured the whole program context. As the ultimate goal of the complexity analysis of an algorithm (or a method) is to find a closed-form expression for the cost (e.g., running time) on the input size, Coppa et al. (Coppa et al., 2012) used curve fitting and curve bounding to generate cost functions.

In line with expectations, the overhead of the Coppa et al. (Coppa et al., 2012) approach is significantly high. Their approach requires an average of 30 times the typical run (the peak was 78.3x). For space requirements, their approach, on average, requires two times the usual space. It is normal to have such high overhead given the fine granularity of collected information.

Coppa et al. (Coppa et al., 2012) state that a single run of their tool using the system under test is mostly sufficient. The key observation they highlight is that the number of distinct RMSs for each method will not increase under the same input. Distinct RMSs are more critical than RMSs with different values because

they expose different paths within the same code block. Nevertheless, we think this is the essential limitation on such approaches (Coppa et al., 2012; Zaparanuks & Hauswirth, 2012), as low distinct RMSs are possible given that developer's inputs usually target functional testing.

Closely related to these two major techniques (Coppa et al., 2012; Zaparanuks & Hauswirth, 2012), Chen et al. (B. Chen et al., 2016) generate cost functions for selected paths. Based on given inputs, they classify paths into high-probability and low-probability paths. The high-probability paths are those that execute under most of the given inputs. Respectively, low-probability paths represent the corner cases of the program that found based on a small number of inputs. Chen et al. (B. Chen et al., 2016) use symbolic execution (Păsăreanu et al., 2013) to classify the given paths and use loops unfolding to ensure the scalability of the technique. The high-probability paths are a representation of the program's normal execution. Understanding the program performance under these cases helps developers understand the state of the program's normal behavior. Low-probability paths, on the other hand, represent the usually untested cases by developers. Highlighting these cases brings developers' attention to unanticipated issues. Given the symbolic inputs and the high-probability and low-probability paths, Chen et al. (B. Chen et al., 2016) generate cost functions to communicate their findings.

Symbolic inputs are efficiently translatable to actual inputs in theory (Visser, Păsăreanu, & Khurshid, 2004). However, given how Chen et al. (B. Chen et al., 2016) treated loops to prevent an explosion on the number of possible paths, interesting performance insights are not explored since the majority of performance issues occur within loops as discussed in Section 2.1.2. Moreover, the inputs generated by the symbolic inputs are not pathological. Meaning they express the

change in the input scale rather than the inputs' structure. Thus, also missing meaningful performance analysis opportunities.

The approaches presented (B. Chen et al., 2016; Coppa et al., 2012; Zaparanuks & Hauswirth, 2012), regardless of other limitations, suffer from the essential issues of ensuring that given inputs are sufficient for driving the analysis tools into interesting performance issues. For techniques such as Coppa et al. (Coppa et al., 2012), this can be mitigated by incorporating the code coverage insight to ensure that the highest number of paths is taken. However, this is beyond the problem of input generation.

2.2.1.2 Inputs as Configurations.

A typical root cause of introducing performance issues is the misuse of off-shelf software (Han & Yu, 2016; Jin et al., 2012; Nistor, Jiang, & Tan, 2013). Developers usually do not clearly understand how to use an API (Application Programming Interface) or APIs behavior changes across different versions of the same system, but no update is applied to their calls. Whether these off-shelf software are libraries or standalone programs, the configurations passed are considered as inputs. From this perspective, few techniques (Siegmund et al., 2012; S. Zhang & Ernst, 2014) analyzed how different configurations (inputs) combinations might influence the application's performance.

Performance issues introduced on applications that were working as expected can be hard to understand for software engineers. These performance issues usually arise by introducing wrong configurations over different versions of the software. Zhang and Ernst (S. Zhang & Ernst, 2014) introduced a recommender system to choose the correct configurations for the desired performance.

To identify and report configuration changes that cause performance issues, Zhang and Ernst (S. Zhang & Ernst, 2014) require two different versions of the same artifacts. Instrumenting the code and using the same configuration, they use the user's usage (test cases or actual usage of the system) to record traces. The instrumentation follows simple techniques that identify predicates on branching control-flows and count their executions. Given traces, Zhang and Ernst (S. Zhang & Ernst, 2014) match predicates of the traces on the old version of the system to the ones from the newer version. They then compute the behavioral deviation for the matched predicates from different versions within a given method. The gained insight indicates how similar or different these two predicates are within different versions. Using thin-slicing, following the dependency between a slicing criterion (e.g., statement initialization) and predicate using only the data flow dependency, to identify the relationship between a given behavioral change and a configuration option, Zhang and Ernst (S. Zhang & Ernst, 2014) recommend which configuration is most likely the cause of the behavioral deviation.

A distinguishable limitation in Zhang and Ernst's (S. Zhang & Ernst, 2014) approach is the necessity of two different versions to measure the influence of configurations. Access to older versions of the same software might sometimes be difficult, if not impossible. However, even if older versions are available, Zhang and Ernst's (S. Zhang & Ernst, 2014) approach does not look into how the given configurations influence the system compared to other configurations for the same version. Thus, they might miss performance optimization opportunities that are actual to the software. To mitigate this Siegmund et al. (Siegmund et al., 2012) proposed an approach that looks at how a user can select the optimal configurations while maintaining the desired performance.

The work presented by Siegmund et al. (Siegmund et al., 2012) target large systems where configurations changes by the user could significantly affect the performance. For example, measuring the performance of a database management system with *indexing* turned on or off can provide helpful insight about the *indexing* effect on performance. The number of such features can be significant in large systems. Moreover, it is hard to predict their effect on performance by users. Even in Siegmund et al. (Siegmund et al., 2012), the number of features can be an obstacle because the number of interaction possibilities is exponential on the number of configurations. They compose the features that cannot be measured in isolation into a single feature to reduce the number of possibilities to mitigate the scalability challenge. In addition, Siegmund et al. (Siegmund et al., 2012) focused on test heuristics to trim the search space. Given these guidelines, Siegmund et al. (Siegmund et al., 2012) predict the system performance and report it to the users.

Such techniques have a different definition of what we considered input. Nevertheless, these are still passive techniques (no new inputs generated).

2.2.1.3 Input Driven Analysis.

The need for input-driven performance analysis was grasped by a few established techniques (Ayala-Rivera et al., 2018; Küstner et al., 2010), but automation is very limited to nonexistent. Such established ideas recognize the importance of the inputs to drive the software engineers' understanding of the software's performance. Moreover, they understand the number of iterations and the deep understanding needed to find inputs that influence performance. Thus, these techniques provided tools that assist in the performance analysis process based on the manually provided inputs.

Küstner et al. (Küstner et al., 2010) present the simplest form of the techniques. Given that inputs highly influence the application under test

performance, Küstner et al. (Küstner et al., 2010) provided a tool that highlights code blocks based on their inputs. There is no automation on the tested inputs. Rather they provide an input based perspective of the application performance. For example, given the selected set of methods a developer would like to understand, the developer defines input ranges (e.g., $x < 5; 5 <= x <= 10; 10 < x$). Using the given ranges of inputs, the tool generates three different profiles for each input range based on the provided test suite. Küstner et al. (Küstner et al., 2010) put much emphasis on the input when analyzing the application under test. However, the needed manual interference by the developers is an obvious limitation.

Ayala-Rivera et al. (Ayala-Rivera et al., 2018) also focus on the workload to assist developers in boosting their productivity. However, compared to Küstner et al. (Küstner et al., 2010), they provide some automation of the workload. Instead of providing only performance feedback and wait for new inputs, Ayala-Rivera et al. (Ayala-Rivera et al., 2018) allow the developers to identify a set of essential input parameters and their characteristics. The given inputs are then automatically stressed (e.g., quadratically increasing an array size for each new execution) based on predefined policies to inspect possible performance issues. Although such an approach might have some automation to manipulate the workload, it does not explore any unanticipated issues by providing scalable inputs, as scalable inputs are not necessarily expressive inputs of performance issues.

2.2.2 Input Generation. Finding a solution that would drive all the presented performance analysis techniques to actual performance issues requires searching the space of all possible special inputs given the whole program context (Korel, 1990). Special inputs are not only stress of some input size (e.g., increasing a size of an array for a sorting algorithm (Burnim et al., 2009)), random generation

of load inputs then passively select the most diverse (P. Zhang et al., 2011) or focus on increasing the coverage of the tests (Cadar et al., 2008). Instead, it is a deeper understanding of the given method or algorithm functionality.

In this section, we walk through distinguishable attempts to generate new special inputs automatically for performance analysis. Distinguishable input generation techniques use machine learning methods (Grechanik et al., 2012), genetic algorithm (Shen et al., 2015), or fuzzing (Lemieux et al., 2018; Petsios et al., 2017) to search for special inputs.

2.2.2.1 Machine Learning Driven. The earliest found technique to use machine learning as a driver to search for special inputs presented by Grechanik et al. (Grechanik et al., 2012). They created a tool called FOREPOST that takes initial inputs and their associated execution times to generate new possible special inputs.

FOREPOST (Grechanik et al., 2012; Luo, 2016a; 2016b; Luo, Poshyvanyk, Nair, & Grechanik, 2016) execute the application under test on a small set of randomly chosen test inputs. Then it infers rules with high precision for selecting test input automatically to drive the application toward possible performance issues. Rules are in a form of `if-then` statements. For example, based on the loaning system evaluated, a rule could be "if inputs `convictedFraud` is `true` and `deadboltInstalled` is `false` then the test case is good." The example indicates that using the given inputs leads to an expensive performance (more computation time); thus, it is a good test case as it exposes performance issues. As input data are clustered into expensive and cheap tests, FOREPOST report methods are marked with expensive test cases. The reported methods are most likely to contribute to a bottleneck.

It is important to understand how FOREPOST obtains performance rules to understand the usage of a machine learning technique. FOREPOST uses the set of values of the application under test as input to an unsupervised machine-learning algorithm. Such input can be represented as $V_{I_1}, \ldots, V_{I_k} \rightarrow T$ where V_{I_m} is the value of the input I_m and $T \in \{Good, Bad\}$. The machine learning classification algorithm learns the model and outputs rules of the form $I_p \odot V_{I_p} \bullet I_q \odot V_{I_q} \bullet \ldots \bullet I_k \odot V_{I_k} \rightarrow T$, where $\odot$ is one of the relational operators and $\bullet$ stands for logical connector `and` or `or`. Such learned rule is sent back to the testing script to automatically collect execution costs and guide the selection of new input data. Repeating the process will partition the input data and generate newly learned rules. The algorithm reaches a high degree of probability of expensive input values if no new rules are learned.

Grechanik et al. (Grechanik et al., 2012) argue that frequently invoked methods, which appear in cheap and expensive test cases, can be of no significance to performance insight. Rather, FOREPOST reports less frequently invoked methods that appear within expensive test cases or have little to no significant impact in cheap test cases. It is clear that such an argument does not always hold as simple examples (e.g., sorting algorithms) could have good and cheap test cases over many invocations and still be of performance significance. However, such observation can be domain-specific (Bergel, Nierstrasz, Renggli, & Ressia, 2011) as Grechanik et al. (Grechanik et al., 2012) mainly evaluate their tool on closed-source loaning application and select only Boolean inputs to manipulate.

Evaluating FOREPOST shows that the technique does generate inputs that worsen the performance. For example, under random testing of *JPetStore*, a widely used java benchmark, it takes an average of 576.7 seconds to execute 125,000

transactions. With FOREPOST, executing the same number of transactions takes an average of 6,494.8 seconds. However, FOREPOST is less efficient in finding bottlenecks; examining FOREPOST's top 30 possible bottlenecks methods for a Renter application results in finding a single performance issue of wasted computations.

We believe that the limitation in ranking interesting performance issues is not a significant one. We argue that reporting bottlenecks should not be examined using input generation techniques (see Section 6.1.1). Instead, it should only feed such data to specifically built profilers (e.g., *gprof*).

Reinforcement learning (another form of machine learning) is adapted to generate test cases (Ahmad et al., 2019; Porres, Ahmad, Rexha, Lafond, & Truscan, 2020). However, there are some known limitations to these techniques. Most notably, their limit to manipulating primitives to test the input size rather than finding an expensive pattern high-level input pattern.

A hopeful advantage in using reinforcement learning is to have the agent generate expensive inputs for x number of preceding versions of the same application. Reinforcement learning (and machine learning in general) are expensive to train. Therefore, a good fit for reinforcement learning is to consider it within a continuous integration environment. Hence, only small training increments that build on top of an established model are required. Established reinforcement learning techniques (Ahmad et al., 2019; Porres et al., 2020) do not demonstrate how they fit in a continuous integration environment.

A known advantage of reinforcement learning is handling a search space with a vast number of parameters. As demonstrated with game training (Mnih et al., 2013; 2015), reinforcement learning can handle a stream of pixel data from

images to learn the position of the game and take actions accordingly. Established reinforcement learning techniques (Ahmad et al., 2019) for input generation demonstrate their work on a small number of parameters (four elements to generate inputs of length less than 10). Real-world applications are only expected to have numerous parameters to permute and will most likely need inputs of more than ten elements to demonstrate an expensive pattern.

A possible cause for the reduced number of elements is the limited feedback Ahmad et al. (Ahmad et al., 2019) can collect. They assume the environment is a black-box. Therefore, they only have access to partial run information (e.g., execution time). However, constraining the problem to be black-box analysis only is unnecessary. The goal of using a profiler is to fix performance issues. Hence, it is safe to assume the availability of the source code and plan for more competencies feedback.

The known successes of reinforcement learning on other complex domains and the established techniques (Ahmad et al., 2019; Porres et al., 2020) demonstrate the applicability of reinforcement learning to search for pathological test cases. However, they do not thoroughly demonstrate an excellent working example according to the limitation described above. Using the successfully tested models of reinforcement learning and the formalizations and techniques that reduce the search scope in software systems, reinforcement learning can have an advantage over other methodologies to generate pathological inputs.

2.2.2.2 Genetic Algorithms Based.

Another distinguished approach for input generation was proposed by Shen et al. (Luo, 2016a; 2016b; Shen et al., 2015). Their definition of the input generation for performance problems is similar to the one given by Grechanik et al. (Grechanik et al., 2012).

However, as we will explain later, their definition lacked a demonstration of generality toward evaluating their approach. Shen et al. (Shen et al., 2015) define the input generation problem as a search and optimization problem. Moreover, they suggest using a genetic algorithm to drive the search task. Shen et al. (Shen et al., 2015) argue that machine-learning-based techniques (Grechanik et al., 2012) fit pattern recognition rather than search and optimization problems. Thus, because the genetic algorithm's core idea is to find new fitter "*individuals*" based on existing ones, they argue it is a good fit for inputs search for performance testing.

Although we think that a good classifier (machine learning algorithm) is suitable for input generation if combined with a sound input selection method for testing, a genetic algorithm is also a good fit if combined with a non-random input selection method. As we will describe it, a significant limitation in the genetic approach is the possibility of falling into less important local-minimas when searching for special inputs.

Shen et al. (Shen et al., 2015) major contributions are on explaining how to represent inputs using genetic algorithm. In the genetic algorithm, they have what is known as an *individual* who is essentially a *chromosome*. *Chromosomes* in their part are made of a set of *genes*. The goal of genetic algorithms is to generate new *individuals* by crossing-over fit *chromosomes*. Calculating the fitness of a *chromosome* is based on a predefined fitness function that considers each *gene*. Figure 6 shows a representation of the *chromosomes* tailored for the input generation problem.

For each set of inputs, Shen et al. (Shen et al., 2015) consider these as *chromosomes*. Within each given *chromosome*, we have a set of *genes* that represent an individual parameter and its value. For example, if a method accepts

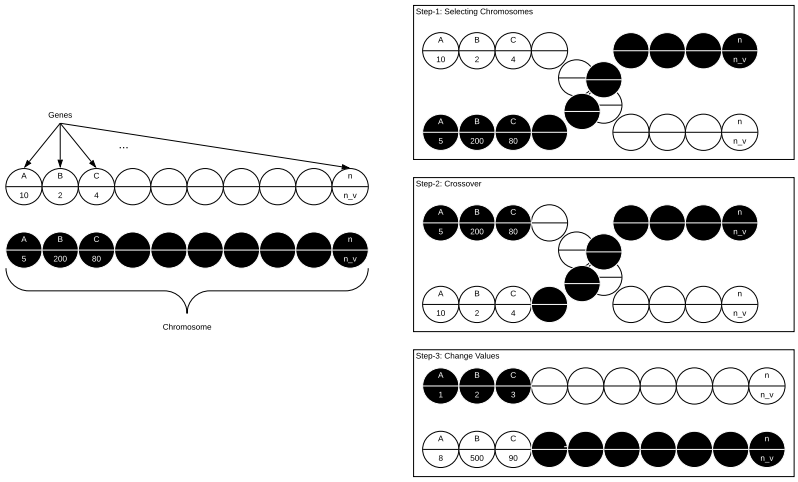


Figure 6. An abstract representation of using genetic algorithms for input generation.

an array `x = [1, 2, 3]` and a boolean `y = True`, then a *chromosomes* is the sequence of *genes* $\{x_1 = 1, x_2 = 2, x_3 = 3, y = True\}$. Using inputs with different values for the same method, the fittest sequences of inputs (i.e., *chromosomes*) are crossed-over to generate a potentially new fitter sequence of inputs.

The fitness function maps the input values to the elapsed execution time. Inputs that maximize the fitness function are fit inputs. Given a run of the application under test based on randomly generated inputs, the fitness function will have few candidate sequences of inputs. As shown in Figure 6, each pair of the candidate sequence of inputs (i.e., *chromosome*) will be crossed over. The crossover phase consists of selecting a subset of the inputs (i.e., a set of *genes*) and exchange them between the sequence of inputs. Finally, for each crossed subset of inputs, the values are randomly changed (authors did not provide details on how they select new values; thus, we assume it is random). In the last step of each iteration, the application under test is rerun using the newly generated sequence of inputs to calculate its fitness.

The presented evaluation by Shen et al. (Shen et al., 2015) does not provide precise results to assess the technique's potential. The evaluation uses URLs as inputs, which does not correlate clearly with how the approach applies to widely available non-web-based applications. Moreover, the experiments highlight the technique's results on injected performance issues. Injecting performance issues does not help in understanding the possible limitation of the technique. Thus, it is hard to draw conclusions about the approach.

Regardless of the evaluation methodology, we can identify some possible limitations. First, the crossover step between two sequences of inputs (i.e., *chromosomes*) by itself does not necessarily generate new inputs. Trying different combinations of the same set of values can lead to some interesting results, but no new inputs are generated. Thus, we fall into the essential issue of the developer's given inputs that does not necessarily cover all the performance possibilities. Second, based on the previous limitation, Shen et al. (Shen et al., 2015) randomly generated new values for each exchange input (i.e., *gene*). This solution does introduce an actual new input. However, because the given genetic algorithm does not calculate how each value (i.e., *gene*) is contributing to the fitness of each sequence of inputs, the new values are not necessarily selected with a high potential of generating new fit sequences of inputs that drives the application's performance toward performance issues. The absence of a link between the newly selected values and the new combinations of inputs is an essential limitation of such approaches.

2.2.2.3 Fuzzing Driven Inputs.

To our knowledge, Lemieux et al. (Lemieux et al., 2018) presented the most general and focused approach to generate input that leads to performance issues and target the comprehensive definition of

performance inputs. They precisely target *pathological inputs* by always fixing the size of the manipulated set of inputs.

Lemieux et al. (Lemieux et al., 2018) use fuzz testing as an engine to drive the input generation. Fuzzing is widely used in functional requirements testing where the application under test is barraged with randomly generated test cases. Fuzzing has also demonstrated a good use for security testing as they typically exposed with totally unexpected test cases. For performance testing Petsios et al. (Petsios et al., 2017) are the first to use a feedback-directed mutational fuzzing to increase the code coverage. Their intuition is to iteratively use evolutionary search techniques to maximize a program execution cost.

As shown in Figure 7, Petsios et al. (Petsios et al., 2017) developed a technique called SlowFuzz that uses a fuzzing engine to generate inputs. In addition, SlowFuzz defines a cost function to rank the inputs based on their execution cost. The example shown explains how the fuzzing algorithm iteratively finds a sorted array that maximizes the cost of executing the given *quicksort* algorithm (i.e., increasing the length of execution paths).

Lemieux et al. (Lemieux et al., 2018) adopt the same methodology, but instead of targeting inputs that only maximize bucketed coverage of the execution cost (e.g., SlowFuzz (Petsios et al., 2017)), they also considered inputs that shows any maximization on path hit rate. SlowFuzz is greedy in that it looks for inputs that maximize the count of edges on the control-flow graph over exploring new paths in favor of achieving a worsen performance in a shorter time. As the search space of any target application is only assumed to be non-convex, *SlowFuzz* is more likely to fall into local-minimas.

```
function quicksort(array):
    /* initialize three arrays to hold
    elements smaller, equal and greater
    than the pivot */
    smaller, equal, greater = [], [], []
    if len(array) <= 1:
        return
    pivot = array[0]
    for x in array:
        if x > pivot:
            greater.append(x)
        else if x == pivot:
            equal.append(x)
        else if x < pivot:
            smaller.append(x)
    quicksort(greater)
    quicksort(smaller)
    array = concat(smaller, equal, greater)
```

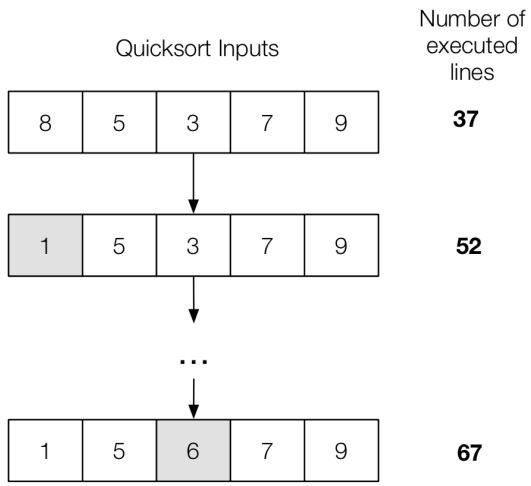


Figure 7. Steps taken by SlowFuzz to find inputs that maximize the execution cost of *quicksort* (Petsios et al., 2017).

Lemieux et al. (Lemieux et al., 2018) implemented their approach in a tool called PerfFuzz. Initializing PerfFuzz requires seed inputs that are known to run the program. The seed inputs are added to a set called `ParentInputs` that maintains known special inputs. The set holds inputs known to either maximize

the bucketed test coverage or maximize the execution cost of the given control-flow graph. Lemieux et al. (Lemieux et al., 2018) argue that such a strategy allows them to avoid local maximums by having a multi-dimensions objective. Remarkably, they argue that it helps them not necessarily finding the global maximum but potentially many different near-global maximums.

If we look at a single iteration of PerfFuzz's algorithm, we find that the algorithm first generates new inputs by randomly permuting the bytes of inputs from the `ParentInputs` that has potential. Bytes permutation simplifies the input scope problem. However, it might not be a good approach for complicated inputs such as widely used data structures. Potential inputs, nevertheless, are the ones from the `ParentInputs` set that maximizes performance value for some cost measurement in the application. The cost definition can differ based on the developer's needs. For example, it could be the number of bytes allocated at `malloc` statement or cache misses. For PerfFuzz they defined *cost* as the execution counts of control-flow graph edges.

All newly found inputs are defined as `ChildInputs`. As PerfFuzz finds a collection of new `ChildInputs`, the application under test is rerun given the new collection. PerfFuzz adds inputs to the `ParentInputs` set if they show a maximization in the bucketed coverage or maximization on any edge hit. The number of inputs in the `ParentInputs` set cannot exceed the number of control-flow graph edges at any given point in time. Thus, PerfFuzz reevaluates all inputs within the set of `ParentInputs` each time a new input shows a performance maximization. The algorithm repeats this process of finding new inputs and running the application against them until it hits a given time threshold (e.g., 1 hour).

Lemieux et al. (Lemieux et al., 2018) evaluate their findings compared with the ones generated by SlowFuzz (Petsios et al., 2017). PerfFuzz outperforms SlowFuzz in finding more pathological inputs. However, an essential issue in fuzzing-based approaches is the time required to generate pathological input. The demonstrated evaluation was based on a 6-hours run for each target application, which are micro to small. Even with the small applications, the results achieved are only possible thanks to the well-engineered AFL fuzzing engine (Zalewski, 2013). Because they generate completely random input, fuzzers usually waste a significant number of computations on trivial or invalid inputs.

Another less critical limitation is the difficulty of understanding the output of the fuzzing engine concerning the performance issue they trigger. For instance, applying *PerfFuzz* on a WordFrequency application would generate results similar to the ones shown in Figure 8. The shown three inputs are all revealing different performance issues. Input (1) in Figure 8 depicts a single long word issue, which maximizes the time taken by the application to compute a hash of the word. The second input (2) in Figure 8 exercises a case where a repeated execution of the method `add_word()` occurs because of the many short words. The last given input (3) presents a case where many hash collisions occur, thus a longer execution time in traversing a linked list.

```
(1) "tvÇ1PFEj??A4A+v!^?^AE!§^?MPttò8dg80ÿ(8mrÿÿÿÿ"
(2) "t t t t i nv t X t 1 9 t l t l t t t t t"
(3) "t <81>v ^?@t <80>!^?@t <80>!t t^Rn t t t t t t t t"
```

Figure 8. Different inputs generated by PerfFuzz (Lemieux et al., 2018) for the WordFrequency application.

All the given inputs in Figure 8 are valid and valuable examples. However, relating them to the root causes of the performance issue requires a deep understanding of the application under test. In addition, as the size of the application grows, the problem worsens.

Despite their limitations, fuzzing solution (PerfFuzz in particular) represent the states of the art techniques in finding pathological input. However, unlike security testing, performance testing usually is affirmed within *known* set of possible inputs. In addition to wasting many computational cost opportunities, the random test case generation will usually not exceed a parsing module of real-world applications. Hence, fuzzers will have more success in revealing performance issues related to validating the input rather than the issue related to the core functionality of the application under test. A solution that searches within the valid set of input using fuzzers or any other search methodology can have more effective results.

2.3 Related Work Summary and Research Opportunities

The need to understand software performance behavior has long been recognized (Ammons et al., 1997; Ball & Larus, 1996; Graham et al., 1982; Larus, 1999). The need for performance satisfactory applications or algorithms has been studied long before the software was written on a large scale. The problem has existed since the formalization of what is known now as the study of algorithmic complexity. Software performance analysis is still an active research area. The level of abstraction at which the solution works, the definition of the performance issues, the diverse nature of software performance requirements, and the decision to fix or only highlight performance issues are a few of the factors that make the problem manifold. Moreover, the inherent habit among the software engineering community

of "fix it later" (Dugan, 2004), makes performance analysis researchers confront a wildly complicated problem. A clear definition of the problem is necessary to narrow down the focus of our work.

Jin et al. (Jin et al., 2012) define *performance issues* as bugs that would enhance the software performance with simple solutions while preserving software functionality. However, such a definition introduces other issues, such as defining a simple solution and how much enhancement is acceptable. We know that the actual measurement is in the cost (monetary value) of having a lousy performing application on production (Kim, Rhee, Lee, Zhang, & Xu, 2016). Unfortunately, if such cost is used to define performance issues, it is most likely too late to discover or fix these issues.

We define *performance issues* as a poor software performance that contradicts the developer's understanding or escapes his/her anticipation. As software evolves by integrating several modules, prior developers' understanding of how each procedure is performing does not necessarily always hold. In addition, usage of out-source libraries introduces a risk of misusing their interfaces (Jin et al., 2012). The workload mismatch between what the developer anticipated and reality is a good indicator of performance issues.

Our definition does not cover all cases in which a performance issue is identified. Less common causes of introducing performance issues beyond the software scope (e.g., user behavior change) might occur. However, we think software engineers' understanding of the code behavior and anticipation of workload at the deployment time is sufficient.

Similar to the performance issue, it is also hard to categorize performance analysis tools based on the published literature. The main objective of a

performance analysis tool is usually to fix performance issues. Some performance analysis tools are designed to take it upon themselves to fix performance issues (Selakovic et al., 2017). Others focus on particular but complex known patterns of issues to highlight them to developers (Dhok & Ramanathan, 2016; Mudduluru & Ramanathan, 2016; Nistor, Song, et al., 2013; Wert et al., 2013; Xiao et al., 2013). Tracking the input to understand their influence on performance is also a different objective for some performance analysis tools (Coppa et al., 2012; Grechanik et al., 2012; Küstner et al., 2010; Xiao et al., 2013).

In addition to the different objectives, there are different targeted environments. Some performance analysis tools are focused on solutions that are only applicable to parallel programs (Curtsinger & Berger, 2015). Others focus on analyzing software entities and their interactions in the environment of distributed systems (Boehme et al., 2016; Herodotou & Babu, 2011; Ofenbeck et al., 2014; Shende & Malony, 2006). Such diversity in the objectives and environments makes it hard to categorize performance analysis tools. More importantly, it makes it harder to compare their effectiveness.

One of the challenges in software performance analysis literature is how to compare techniques' effectiveness concerning performance. Due to diverse objectives (e.g., targeting performance anti-patterns (Wert et al., 2013), considering configuration as inputs (S. Zhang & Ernst, 2014), and improving the understandability of the results (Della Toffola et al., 2015)), each solution would emphasize its analysis goal more during evaluation. For example, Curtsinger and Berger (Curtsinger & Berger, 2015) argue that the performance improvement opportunities they discover are far more effective than those found by *gprof* (Graham et al., 1982) for the given benchmark. However, it is known that profiling

parallel program was never an objective for *gprof* (Graham et al., 1982) as it was for Curtsinger and Berger (Curtsinger & Berger, 2015). By fixing the highly ranked hotspots, the overall application speed-up could be seen as a good measurement of the technique's effectiveness. Nevertheless, such an indicator can be biased as the fix is highly dependent on the developer's experience and understanding of the applications. The technique's added value to developers (e.g., ease of use, root cause understandability, and other feedbacks) could be a good measurement but hard to capture.

For software engineers, a valuable performance analysis tool would provide fine-grained details efficiently (Ball & Larus, 1996), searches for new and unanticipated behaviors (Lemieux et al., 2018), and identify the root cause of the performance behavior (Selakovic et al., 2017). Such challenging characteristics of a performance analysis tool would help software engineers assert their understanding of their written software or reveal an unanticipated performance issue.

Collecting very detailed traces of an application run is costly. There is a trade-off between how detailed the information provided by a performance analysis tool and how acceptable the overhead is. Details can be of different types. For example, effectively collecting the number of times a path is taken within an application among all possible paths is one type (Ball & Larus, 1996). Another type would be identifying possible inputs of all executed methods (Coppa et al., 2012). Such fine-grained information is usually associated with a high overhead cost that could preclude their adoption in the real world.

In addition to the overhead trade-off, existing performance analysis tools rarely attempt to discover unanticipated worst-case scenarios. For dynamic analysis techniques, the primary cause of such limitation is the dependency on the

developer's written unit tests to drive the analysis of a given application. Such unit tests are usually written to ensure the preservation of the application's functional requirements. Thus, the result of using such unit tests can rarely lead to interesting performance observations. Different inputs alone could have a significant effect on analysis outcomes. There has been some effort to study the effect of the inputs on the performance analysis process (B. Chen et al., 2016; Coppa et al., 2012; Küstner et al., 2010; Xiao et al., 2013), but these either did not attempt to generate new and interesting inputs to drive the test or were limited to unique input generation and permutation cases.

Recent effort explored possible solutions to generating inputs for the goal of performance analysis (Grechanik et al., 2012; Lemieux et al., 2018; Petsios et al., 2017; Shen et al., 2015). These techniques differ on how they define the issue of pathalogical inputs. However, the work by Lemieux et al. (Lemieux et al., 2018) is that first that actually address such issue broadly and attempt to generate inputs based on fuzzing techniques. The outcomes of their evaluation shows promising results that indicates minimal improvements could lead to significant performance insight about the application under test using the same existing profilers.

Moreover, existing performance analysis tools focus on locating possible performance issues but not communicating these to the developer in an understandable use case or architectural abstraction (Z. Chen et al., 2018). Result understandability is a significant criterion that performance analysis tools should consider. Misunderstanding a tool result would lead to wasted optimization opportunities (Nistor et al., 2015). Usually, there is a significant trade-off between how comprehensive a performance tool is and the daptation of its results due to difficulties in understanding it (Nistor et al., 2015; Selakovic et al., 2017).

The leading fuzzing-based test case generation techniques (Lemieux et al., 2018) suffer from multiple limitations. First, it randomly generates input for performance testing, which does not pass the parsing module of the target application in most instances. Second, it wastes a significant amount of computational budget by generating invalid inputs. Third, fuzzers' produced test cases are most likely to be challenging to comprehend, making the result adaptation harder.

To this extent, we aim to use grammar obtained from the target application own parser or documentation to drive the search of expensive inputs toward the core functionality of target applications. Furthermore, using a grammar-based search engine will result in test cases that match the developer expectations of the application's input. In addition, we think adapting a more structured search technique such as Monte Carlo tree search or reinforcement learning would waste less computational effort; thus, expediting the search process.

In the following chapters, we present a necessary background for formalizing each technique and present our contribution of each technique in separate chapters and discuss each one.

CHAPTER III

BACKGROUND

The production rules of a context-free grammar (CFG, Section 3.1) can produce different texts depending on which production rules are chosen at each step. These choices can be seen as defining a search space, with the start symbol at the root of a search tree. Our primary contribution is adapting Monte Carlo tree search (MCTS, Section 3.2) as the search technique to explore portions of the search space defined by a CFG. Monte Carlo tree search uses random probes, called *rollouts*, in lieu of a static heuristic to determine which choices in the tree merit more search effort. An alternative would be to use reinforcement learning (RL) techniques to *learn* a robust heuristic. We also explored RL techniques to guide test data generation but found it less effective. The fundamentals of RL are given in Section 3.3.

3.1 Context-Free Grammar

Context-free grammars (CFGs) are a natural and common choice for describing text with rich, recursive structure, including not only programming language source code but also configurations and instructions for many applications. A CFG can be specified as a set of rewrite rules of the form $A \rightarrow a$, where A is a *non-terminal* symbol and a is a *phrase*, a sequence of zero or more non-terminal and terminal symbols. For our purposes the terminal symbols are string literals. We will refer to terminal symbols as *literals* to avoid confusion with terminal nodes in a search tree, and we will treat a sequence of literals as equivalent to their concatenation.

Valid inputs for an application comprise a formal language, but complete rules to accept all and only valid inputs, including semantic constraints, may

not be expressible by any description less complex than the application itself. A Context-Free language is far more limited, expressing only limited syntactic rules of well-formedness. However, a Context-Free language can be described simply and compactly by grammar, and while the set of sentences generated by the grammar may be a substantial superset of the valid inputs, it is far smaller than the inputs generated by blind mutation.

Formally, a context free grammar $\mathbb{G}$ is a tuple (N, L, R, S). N and L are finite sets of non-terminal symbols and literals, respectively. R is a finite set of production rules in the form $A \rightarrow a$ where $A \in N$ and $a \in N \cup L$. Finally, S is the *start symbol* of the grammar where $S \in N$. Any derived sentences from $\mathbb{G}$ must start from S as a distinguished non-terminal.

⟨*expr*⟩ ::= '(' ⟨*expr*⟩ ')' ⟨*pow*⟩ | ⟨*expr*⟩ ⟨*op*⟩ ⟨*expr*⟩ | ⟨*digit*⟩
⟨*pow*⟩ ::= '^' ⟨*digit*⟩ | /* empty */
⟨*op*⟩ ::= '+' | '-' | '*' | '/'
⟨*digit*⟩ ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

Grammar 1. A Context-Free Grammar for a simple expression solver.

The grammar in Grammar 1 is a concise example of CFG written in BNF notation. To simplify notation we always consider the left-hand-side symbol of the first production rule as the starting symbol of the grammar (i.e., S). Each generated input can be described as a sequence of steps (derivations) in which a non-terminal symbol is replaced by a sequence of non-terminal and literals. For example, the production rule ⟨*op*⟩ ::= '+' | '-' | '*' | '/' from Grammar 1 can be expanded to '+', '-', '*', or '/'. In derivations, we evaluate left-most non-terminal elements one by one until all non-terminals are consumed. To show a working

example, consider generating the sequence `5+2`. Using Grammar 1, the derivation steps would be as the following:

$\langle expr \rangle \rightarrow \langle expr \rangle\ \langle op \rangle\ \langle expr \rangle$

$\rightarrow \langle digit \rangle\ \langle op \rangle\ \langle expr \rangle$

$\rightarrow$ `5` $\langle op \rangle\ \langle expr \rangle$

$\rightarrow$ `5+` $\langle digit \rangle$

$\rightarrow$ `5+2`

The main advantage of using raw seed inputs with fuzzing over a CFG to drive the input generation is the ease of use. Seed inputs are usually easy to create if not readily available. Thus, developers would not need to construct a grammar to use an approach such as PerfFuzz (Lemieux et al., 2018). There have been several attempts to derive CFG from seed inputs (Bastani, Sharma, Aiken, & Liang, 2017; Kulkarni, Lemieux, & Sen, 2021; Wu et al., 2019) to achieve such a goal. We study the applicability of using the most prominent approach (Glade (Bastani et al., 2017)). However, we do not attempt to synthesize grammar ourselves as this is a different research field.

3.2 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is a heuristic tree-based search algorithm (Baier & Drake, 2010; Browne et al., 2012; Chaslot, Bakkes, Szita, & Spronck, 2008; Coulom, 2006; Kocsis & Szepesvári, 2006; Perez, Samothrakis, & Lucas, 2014; Schadd, Winands, van den Herik, Chaslot, & Uiterwijk, 2008). It construct a search tree based on random sampling. The MCTS algorithm has proven useful in many

different sequential decision making Artificial-Intelligence domains. Most notably, MCTS is used in gaming problems.

As the size of a problem is increasingly large, MCTS provides a effective strategy to asymmetrically explore the problem space with emphasis on potentially good decision sequences. MCTS's goal is to not samples every possible decision-sequence but to increase the confidence of making a good decision given a limited computational budget.

To formally explain how MCTS works, we formalize a problem as a Markov Decision Processes (MDP) problem. MDP gives us a way to formalize sequential decision making. Given some environment and an agent, in an MDP we have a set of states S where s_0 is the initial state, a set of actions A, a transition model $T(s, a, s')$ that dictates the probability of reaching state s' if action a is taken on state s, and a rewards function R. As the problem is sequential, at each time step $t = 1, 2, 3, ...n$, the agent use the current state $s_t \in S$ to select a valid action $a_t \in A$ that produces the state-action pair (s_t, a_t). As the time increments (time is the number of steps taken), the environment transition to the next state $s_{t+1} \in S$, and the agent receives a reward $r_{t+1} \in R$ based on its last action a_t.

At each state $s_t \in S$ the agent takes a valid action $a_t \in A$ to form the decision pair (s_t, a_t). The goal for the agent is to find a *policy* that maps state-action pairs from the initial state s_0 to a terminal state $s_{terminal}$ that maximize the expected return $Q(s, a)$.

$$UCT = \overline{X_i} + C\sqrt{\frac{\ln N_i}{n_i}}$$

Most commonly, and in our case, the UCT (Upper Confidence bound apply to Tree) is used to define $Q(s, a)$. Given the transition model T, the expected

return $Q(s, a)$ of the action a that takes us from state s to state s' is found using the UCT formula above. The variable $\overline{X_i}$ defines the average rewards observed from state s'. N_i represent the number visits to state s. Similarly, n_i represent the number of visits to state s' (the child state). Finally, the variable C is a constant that is used to tune exploration vs. exploitation. A value between [0, 2] is commonly used in the MCTS literature. A lower C value will lead to exploiting the tree more where a higher C value will lead to more exploration of the tree.

Now that the basics are formalized, we describe the MCTS algorithm. MCTS build a search tree iteratively until a defined computational budget is consumed. Building the search tree involves four major steps; *selection*, *expansion*, *simulation*, and *backpropagation*. For each iteration in building the search tree the four steps are defined as the following:

- *Selection*: From root node in the tree s_0 select the *best* child according to the UCT formula until you reach a terminal or expandable node. A node is *expandable* if it was visited before but it is not a terminal node (has possible children).

- *Expansion*: Populate all the children from the current node following the TreePolicy where each child node represent a valid state-action pair (s_t, a_t). Then move to one of the populated children (randomly or following some policy as all children have no valid $Q(s, a)$ yet).

- *Simulation*: From the current node, select valid actions according to TreePolicy all the way to a terminal state. These actions are not populated. This step is also known as a *rollout*.

– *Backpropagation*: Given the reward from a *simulation* or a known terminal state back-propagate the reward following the node's ancestors. At each step from a node to its parent the reward is added to the accumulated known rewards and the number of visits is increased by 1.

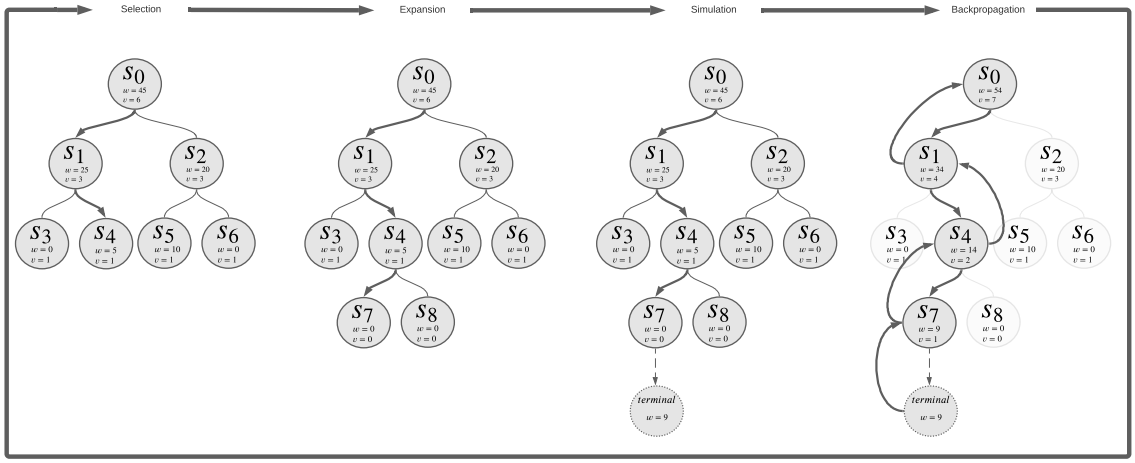


Figure 9. An illustration of the basic steps (*selection*, *expansion*, *simulation*, and *backpropagation*) performed in each MCTS iteration step.

The illustration in Figure 9 provide a working example of how an iteration could go. For each node, the variables w and v are the accumulative wins and the number of visits respectively. In the first step the algorithm selected nodes with the highest UCT value. Second, it Expand the node S_4 as it was visited at least once before. Third, run a simulation from one of S_4 new children (selected randomly) to reach a terminal node and collect reward information. Finally, the value obtained from the simulation is back-propagated through the ancestors of S_7. Note how nodes within a simulation step are not added to the search tree.

A node expansion typically occurs after one visit to some non-terminal node. However, to save memory and increase the probability of the node effectiveness before expanding it, the MCTS algorithm can define a visits threshold for

expansion. The larger the threshold, the more visits will be needed to a node before expanding it. Thus, more rollouts from the same node. Such a strategy reduces the number of nodes created within the search tree and increases the potential of the populated ones.

3.3 Reinforcement Learning

Reinforcement learning is an iterative approach to train agents to do *good* in some environments by providing them with rewards given each action the agent takes within the environment. The process in Figure 10 illustrates the agent interaction with the environment. An environment must have a well-defined state structure to communicate and a set of actions it can accept. An agent interacts with the environment by selecting an action and passing it to the environment. Given the current state of the environment and the agent's action, the environment acts and passes the new state back to the agent along with the reward based on the last given action. The agent's goal is to maximize the reward by learning which action is likely to return the highest rewards on a given state.

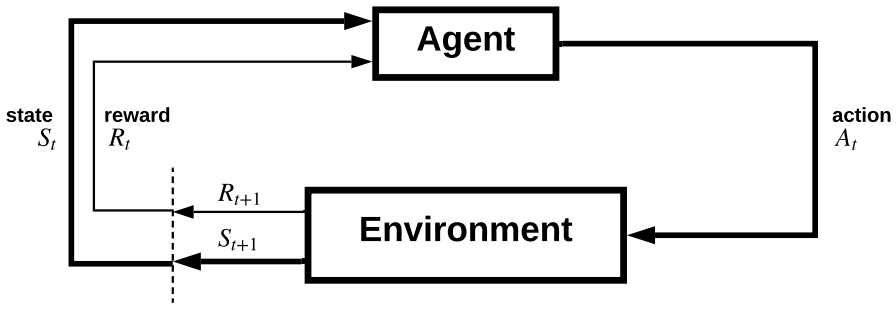


Figure 10. The agent-environment interaction in a reinforcement learning algorithm. The figure is redrawn based on illustration seen on (Sutton & Barto, 2018)

In the following section (Section 3.3.1), we will go over the fundamentals of reinforcement learning. Next, (Section 3.3.2), we briefly introduce the state of the art reinforcement algorithm on deep neural network and briefly discuss some areas where it was applied successfully.

3.3.1 Reinforcement Learning Fundamentals. To formally explain how reinforcement learning works, we need to break the learning process into different parts. The key to any problem to work for a reinforcement learning algorithm is to be formalized as a Markov Decision Processes (MDP) problem. MDP gives us a way to formalize sequential decision making, which is the basis for reinforcement learning. Given some environment and an agent, in an MDP we have a set of states S, a set of actions A, and a set of rewards R. As the problem is sequential, at each time step $t = 1, 2, 3, ...n$, the agent use the current state $s_t \in S$ to select an action $a_t \in A$ that produces the state-action pair (s_t, a_t). As the time increments (time is the number of steps taken), the environment transition to the next state $s_{t+1} \in S$, and the agent receives a reward $r_{t+1} \in R$ based on its last action a_t.

The agent goal in such an environment is to maximize the *expected return* from all state-action pairs (S_t, A_t). The expected return is the sum of all future rewards. Formally, the expected return G at time t is defined as:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T$$

The former expected return definition works with episodic tasks[1]. In the case of continuing tasks (e.g., learning to adjust a thermometer of a room), the final time step's T notion does not apply as this time step would go to infinity (∞).

[1] Episodic tasks are ones with terminal states.

Therefore, the definition of *expected return* needs to be adjusted with a *discount rate* to account for some tasks' continuous notion. Hence the goal becomes to maximize the discounted return of rewards instead of expected rewards' return. The *expected return* formula can be adjusted by choosing a value between 0 and 1 for the discount rate γ to make the agent cares more about immediate future return rather than returns far in the future.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Given the goal and the formalization of the problems, what the agent learns, is a policy of selecting the states' actions. Understating the policy effectiveness requires introducing the notion of value-function. A policy π then is a function that calculates the probability of selecting an action given a state $\pi(a|s)$ at time t. Furthermore, the value-function of a state-action pair tells us *how good* is any given action from the given state. We can find *how good* an action is from a given state by calculating its expected return $\mathbb{E}$ (using the discounted return formula described above).

The action-value function for policy π can be written as q_π. Therefore, the action-value function for selecting action a in state s under π (i.e., $q_\pi(s, a)$) is the expected return starting from state s at time t, taking action a and following policy π thereafter.

$$q_\pi(s, a) = \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right]$$

Finding a policy is not hard. However, finding the optimal policy that yields the maximum expected return is. The policy π is considered better than or the

same as policy π' if the expected return from π is greater than or equal to the expected return of π' for all state-action pairs. There is always at least one policy that is better than or equal to all other policies; that is the *optimal policy* π_*. The optimal policy has an associated optimal action-value function q_*.

$$q_*(s,a) = \max_{\pi} q_\pi(s,a) \forall s \in S, a \in A$$

The learning strategy and the definition of optimality described above are the basic blocks of formalizing reinforcement learning. To cover the complete details of the learning process, we have to understand how expected-return ($q_\pi(s,a)$) values are tracked. Also, how the policies are compared to find optimality.

3.3.2 Q-Learning & Deep Neural Networks. In its simple structure, reinforcement learning uses what is called Q-Learning or Q-Table. Q-Learning algorithm stores the Q-Values mapping between all states and actions and iteratively update these Q-Values given Bellman optimality equation to find the optimal policy. To update the Q-Values, we need to introduce the notion of learning rate α. A learning rate is a number between 0 and 1 that set the percentage of how much the agent should keep from what it learned *before* versus what it knows *now*. Hence, to calculate a new q value, we apply the following equation on each encountered Q-Value.

$$q_{new}(s,a) = (1-\alpha)\ q_{old}(s,a) + \alpha \left(R_{t+1} + \gamma \max_{a'} q(s',a') \right)$$

In the previous section, we always stated that our goal is to maximize expected rewards. Even though this is our goal, such greediness could lead the

agent to fall into some local minima[2] more often than we would like. Therefore in most reinforcement learning algorithms, we use the *epsilon-greedy strategy* to find a balance between exploration and exploitation of actions. The ϵ value determines whether to select an action based on the Q-Values in hand (*exploitation*) or randomly (*exploration*). Reinforcement algorithms use a dynamic epsilon strategy that would select more random actions at the beginning of the training, then the rate decay as the training moves toward the end of the number of set episodes.

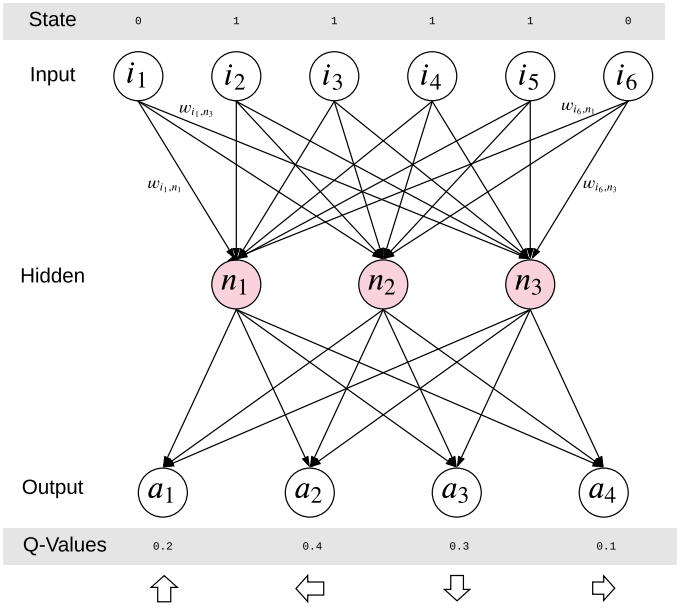


Figure 11. An illustration of a simple neural network. This is not similar to the one explained in Mnih et al. (Mnih et al., 2013). In the Atari paper, they use convolutional neural network as they learn from games screenshots. The illustration here shows the basics of how neural network in general can be used for reinforcement learning. In practice they are much deeper and wider and might involve special components.

The Q-Learning model that we explained is easy to understand. However, the model does not scale to more extensive problems where the number of possible

[2]For any mathematical function, local minima are the values where the functions reached the lowest possible value at sub-space, but this low value is not the lowest value at any point of the function (i.e., global minima).

states and actions is vast. Mnih et al. (Mnih et al., 2013) were the first to introduce the use of Deep Q-Networks (DQN) for reinforcement learning to use the network as an estimator of the Q-Values. Having the deep neural network where the input is the state and the output are the action values for the given state (see Figure 11), Mnih et al. (Mnih et al., 2013) were able to train an agent to play Atari games learning from pixel values as a state and interacting with the environment with the available actions (e.g., *right*, *left*, or *jump*) to reach the human-level expertise.

This basic idea of interaction between the agent and the environment has proven powerful when integrated with the state of the art DQN on different domains. Reinforcement learning techniques have demonstrated effectiveness on complex problems such as playing video games (Lillicrap et al., 2015; Mnih et al., 2016; 2013; 2015; van Hasselt, Guez, & Silver, 2015), text modeling (Hellendoorn & Devanbu, 2017; Kalchbrenner, Grefenstette, & Blunsom, 2014), and image classification (Krizhevsky, Sutskever, & Hinton, 2017). Therefore, it is reasonable to examine reinforcement learning applicability on finding all possible pathological inputs that maximize the execution cost of a given target application.

CHAPTER IV

TreeLine: FINDING SLOW INPUTS FASTER WITH MONTE CARLO TREE SEARCH

In this chapter, we explain our approach to composing inputs for performance testing. We first explain our methodology and detail each contribution (Section 4.1). In Section 4.2, we evaluate TreeLine in comparison with the state-of-the-art input generation fuzzer. We also conduct evaluations that address the significance of the proposed enhancement on top of the Monte Carlo Tree Search (MCTS). At the end of the chapter (Section 4.3), we discuss TreeLine's limitations and possible future work.

The content in this chapter is a result of collaboration with co-author (Michal Young) and is not published yet. Ziyad Alsaeed is the primary author of this work and responsible for conducting all the presented analyses.

4.1 Approach

4.1.1 Overview. TreeLine uses context-free grammar (CFG), as defined in Section 3.1, as a base for its input syntheses. The application's CFG can be provided by the developer or synthesized from raw seed inputs using tools such as Glade (Bastani et al., 2017). We focus on user-defined grammar obtained from program parser or documentation. We discuss the suitability of synthesizer-based grammar later in the chapter.

We construct a tree-based input generator where the starting symbol of the grammar is the root of the tree (Section 4.1.4), and the derivations from the root are constrained with a budget (Section 4.1.3). Using MCTS algorithm (Section 4.1.5), TreeLine performs rollouts and expansions on the derivation tree, leading to more expensive inputs. The feedback of executed inputs is

gathered from the target application (Section 4.1.2). However, we modulate the feedback according to the target application cost range to compensate skewed feedback (Section 4.1.5.2). Also, we enhance MCTS by favoring coverage-increasing nodes (Section 4.1.5.1) and avoiding excessive exploration of visited paths (Section 4.1.5.4).

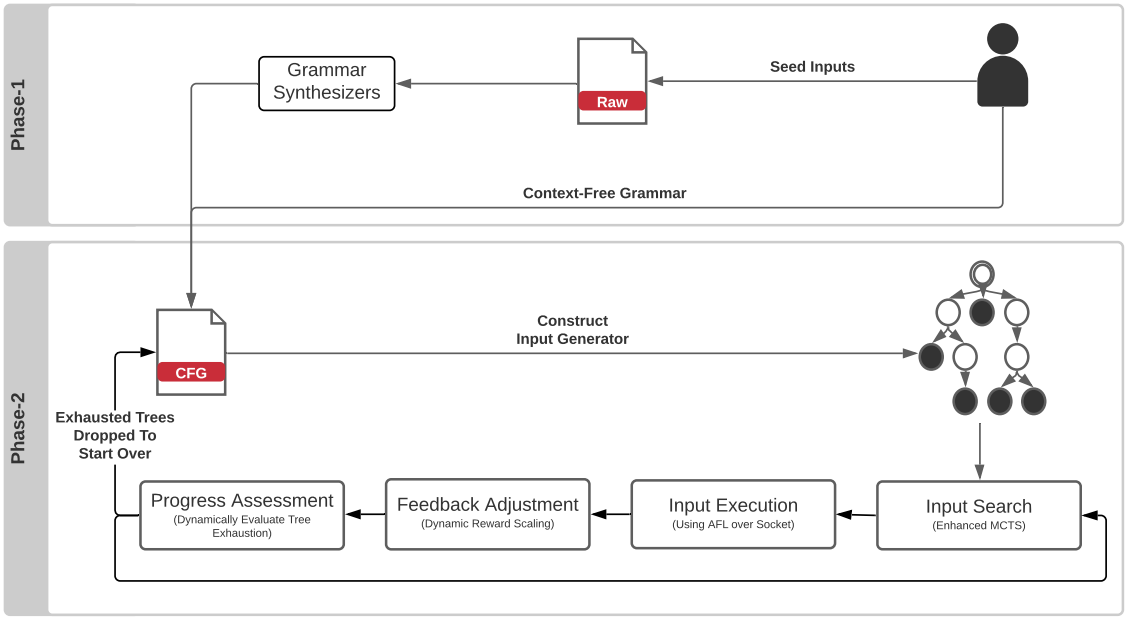


Figure 12. TreeLine's high-level overview. Phase-1 involves manual preparation of seed inputs or grammars. Phase-2 illustrates the automated steps TreeLine go through to search for expensive inputs.

Figure 12 illustrates the high-level process of TreeLine. We split the process into two phases. Although we do some minor enhancements to automatically synthesized grammars in Phase-1, most of our contribution is in the processes shown in Phase-2.

In Phase-2, TreeLine will first parse the CFG into a tree-based input generator. Using the input generator, TreeLine will keep searching for expensive input until it consumes the allowed computation budget (e.g., defined timeout).

TreeLine will search for new expensive input in each search iteration following the enhanced MCTS algorithm we describe in Section 4.1.5. The feedback for running the newly found inputs is obtained using AFL. Our reward method dynamically adjust the feedback to work for MCTS balancing. The rewards are sent to the search tree for back-propagation. At the end of each cycle, TreeLine dynamically assess the exhaustion of the current search tree. If the tree is exhausted, TreeLine drop the tree and construct a new one based on the same grammar to take advantage of the more cultivated rewards. In each search iteration, we keep track of expensive input for final reporting.

4.1.2 Target Application Instrumentation & Run Tracking.

Instrumentation tools like AFL's (Zalewski, 2013) work best with program source code, although instrumentation of binaries is also possible. However, we consider the application's source code for our experiments as it is reasonable to assume that source code is available. A testing tool such as searching for expensive inputs is used to fix performance issues. Using a binary version of the target application may only simplify the integration process with the testing tool. However, the source code would require an inspection to validate the tool's findings and apply fixes.

We rely on AFL to instrument the target application considering the effort put into implementing it over the years. However, we only use the instrumentation and target-application runner and tracker of the tool. We do not use any of the fuzzing tactics that are the core of AFL.

In principle, we can define the cost component with different desired bases (e.g., cache hits, memory-operations, or wall-time), but in our evaluation we use control-flow graph edge hits as our cost component for performance. The control-flow graph is easy to adapt as it is how AFL define cost component. Although

not the only performance indicator, edge hits are a reliable and cheap component to measure. Unlike time, if the target application is deterministic, we expect the input's cost to be the same across different environments and loads of the system. Also, it facilitates a good ground for comparison with other fuzzing tools.

AFL tracks the target application coverage by counting the control-flow graph edges hit rate (an illustration is given in Figure 5 from the Related Work chapter 2.2). Therefore, it is different from well-known profilers (Graham et al., 1982), which count blocks. AFL collects edge hits information each time the target application is executed with new inputs. The edge hits rate forms a base for different possible insights. For example, the total edge hits describe the cost of executing the whole application given some input. And the value of each edge describes the cost of exercising a particular component within the application.

Focusing on each edge in the control-flow graph, one can form two different insights. AFL initial design tracks different states of coverage in which the edges can be categorized. An edge that gets hit for the first time or one that unlocks a bucket of hit ranges (hit 1 time, 2 times, 3 times, 4–7 times, 8–15 times, 16–31 times, 32–127 times, or 128–255 times) increases the coverage. Such feedback is a broad interpretation of the test case coverage. We refer to such feedback as NewCov. An input will exercise a NewCov on the target application if some edge unlocks a new bucket of hits compared to all previously observed inputs.

Another edge-specific insight is based on the granular change to hits for each edge. PerfFuzz (Lemieux et al., 2018) introduced unsummarized feedback by taking advantage of a more precise tracking of any maximizing change in hits to each edge. Instead of categorizing the hits into predefined buckets, PerfFuzz tracks the exact number of edge hits. An input that shows an increase in the

number of hits to some edge in the control-flow graph compared to all previously executed inputs exercises NewMax feedback.

Although the NewCov and NewMax tracking seems to be serving the same purpose, they are different. NewMax tracks any slight increase to hits of a given edge. On the other hand, NewCov tracks different states of coverage, and can register new coverage by decreasing as well as increasing the count of edge hits. Thus, after few iterations, inputs can increase the max hit to an edge with no new coverage and vice versa.

Overall feedback can be drawn from the whole control-flow graph map. From AFL, we also track the new change in the total cost of executing an input. The NewCost is exercised if and only if the sum of all edge hits from the control-flow graph is larger than the last known max sum. Similar to the case with NewCov and NewMax it is possible to have an input that maximizes the NewCost but has no NewCov or NewMax. Our goal in the search tree is to maximize the cost.

With each input run, AFL will always return the feedback elements NewCov, NewMax, and NewCost. We save and favor inputs for which any of the given feedback elements are true.

4.1.3 Cost & Budget. Derivation lengths from CFGs with recursive productions are unbounded (refer to Section 3.1 for fundamentals of CFG). Thus, the depth of the search tree will be unbounded. Also, our definition of pathological inputs dictates that all inputs in search space must be of bounded size, because we want to find a short input that demonstrates a pattern that triggers slow execution, rather than the possibility of slowing an application down with enormous inputs.

Therefore, we introduce the notion of *cost* to grammar and *budget* to tree search to bound the generated input length and tree size.

A cost is associated with each symbol in the grammar $s \in N \cup L$. To constrain the derivation length, we calculate the minimum possible cost based on the literals a symbol $s \in N$ can lead to. Therefore, we have a minimum cost associated with each symbol $s \in N \cup L$.

$$\exists\ \textsc{MinCost}(s), \forall s \in N \cup L$$

The base for defining the cost in any grammar is the literals set L. We can look at the cost of literals $l \in L$ as a token; thus, it costs 1. However, we also can define the minimum cost as the number of characters in that literal symbol. For example, the symbol $\langle B \rangle$ from the production rule $\langle B \rangle$ ::= 'bb' | 'bbb' can have $\textsc{MinCost}(\langle B \rangle) = 1$ considering a token-based cost or $\textsc{MinCost}(\langle B \rangle) = 2$ considering a character-based cost. Because the closely related work for input search (Lemieux et al., 2018; Petsios et al., 2017) adopts a byte-based approach that is even more granular than characters, we adopt a byte-based cost across all experiments unless stated otherwise.

We strongly believe adopting a token-based definition of cost is more suitable for MCTS-based search techniques going forward. Defining the cost of a symbol in the grammar based on the tokens is consistent with the definition of pathological inputs. Real world application have predefined keywords for their inputs. Treating each keywords equally regardless of its size in term of character count is consistent with inputs' length constraint. Although resulting inputs might have different number of characters but they would always have the same number of used tokens. Moreover, using character or byte-based costs could lead to the

search algorithm favoring one keyword over the other due to size only (e.g., `graph` is shorter than `digraph` for graph layout applications). Shorter keywords will allow the accommodation of more other characters. Therefore, shorter keywords are favored due to size. A token-based cost will regularize the importance of keywords.

Given that each literal symbol has a defined minimum cost, we can inductively define a minimum cost for all symbols $s \in N \cup L$. Any non-terminal symbol will have a MinCost based on its options.

$$\text{MinCost}(A|B) = \min(\text{MinCost}(A), \text{MinCost}(B))$$

$$\text{MinCost}(AB) = \text{MinCost}(A) + \text{MinCost}(B)$$

Defining a cost for each symbol allows a budget constraint on how many literals can be used to form inputs. Thus, grammar derivations are bounded. For each derivation step, the options available are reduced in consideration of the remaining budget at that step. For example, with a budget of 2, evaluating the symbol $\langle B \rangle$ based on the production rule $\langle B \rangle$ ::= 'bb' | 'bbb', then only 'bb' is available as an option given that the cost of 'bbb' is larger than the remaining budget. We elaborate on the mechanics of the derivation with budget constraints when we describe our InputGenerator (Section 4.1.4).

Using the cost and budget defined above, we ensure a bounded number of derivations for regular grammar. However, if a grammar has possible cyclic and cost-less (i.e., MinCost=0) symbols, then our defined cost and budget method will not bound the derivation steps. Although it is possible to form such grammars, we found them uncommon with hand-crafted ones, found in documentation, or inferred from the application parser. Nevertheless, a mitigation can be applied by

transforming a cyclic grammar into other acyclic forms like the Greibach Normal Form (Greibach, 1965; Hopcroft & Ullman, 1969).

4.1.4 Input Generator. We described the funcdmenetal of MCTS in Section 3.2. For the CFG to work with MCTS, we need to represent the grammar as a tree to generate inputs. The same generator defines the TreePolicy that enforces the constraints on expansions of the search tree. Therefore, our InputGenerator is a tree-based generator that uses the grammar starting symbol $S \in \mathbb{G}$ to construct a root node n of a search tree.

In relation to InputGenerator, a node n in the search tree has six relevant properties *parent*, *symbol*, *budget*, *children*, *text*, and *stack*. Each property is described as follows:

- $parent(n)$: A pointer to the node's parent.
- $symbol(n)$: The current symbol $s \in N$ from $\mathbb{G}$ we are evaluating. In derivation steps, this is the next non-terminal we are evaluating.
- $budget(n)$: The budget passed from the node's parent.
- $children(n)$: A set of all valid grammar options from the current *symbol* given its production rule and $budget(n)$.
- $text(n)$: Is the produced literals up to this node. This is an empty string in the case of the root node.
- $stack(n)$: Is the set of derived and non-resolved symbols $s \in N \cup L$ so far based on the production choices.

To illustrate how the InputGenerator works we use a simple but representative grammar as shown in Grammar 2 with a small budget $\mathbb{B} = 3$.

⟨*rcompound*⟩ ::= ⟨*edgeop*⟩ ⟨*simple*⟩ ⟨*rcompound*⟩ | /* empty */

⟨*simple*⟩ ::= ⟨*nodelist*⟩ | ⟨*subgraph*⟩

⟨*edgeop*⟩ ::= '->' | /* empty */

⟨*nodelist*⟩ ::= ⟨*node*⟩ | ⟨*nodelist*⟩ ',' ⟨*node*⟩

⟨*node*⟩ ::= ⟨*atom*⟩ | ⟨*atom*⟩ ':' ⟨*port*⟩

⟨*port*⟩ ::= 'n' | 'ne' | 'e' | 'se' | 's' | 'sw' | 'w' | 'nw' | 'c' | '_'

⟨*atom*⟩ ::= `[a-zA-Z0-9_]`

Grammar 2. Excerpt of a grammar transcribed from parser source code found in the `graphviz` source code repository. The full grammar can be found in the Appendix (Grammar 7)

Calculating the minimum cost for each symbol $s \in N$ we get a MinCost of zero for ⟨*rcompound*⟩ and ⟨*edgeop*⟩ as both lead to an empty case and a MinCost of one for all other symbols. A summary of all possible derivation based on the given budget and costs is shown in Figure 13.

The budget on the root node is $budget(n_{root}) = 3$. As the root node is the first node in the derivation we adjust its budget based on its minimum cost compared to the passed budget ($budget(n_{root}) = PassedBudget -$ MinCost($symbol(n_{root})$). For all other nodes, the *budget* is exactly the value passed from the parent node as the value is adjusted with option selection. At each node n, the set of *children* has the options that adhere to the ValidOptions equation below. Thus, from the root node n_{root} all options are valid because for each option $\alpha \in symbol(n_{root})$ the MinCost(α) is <= to $budget +$ MinCost(⟨*rcompound*⟩).

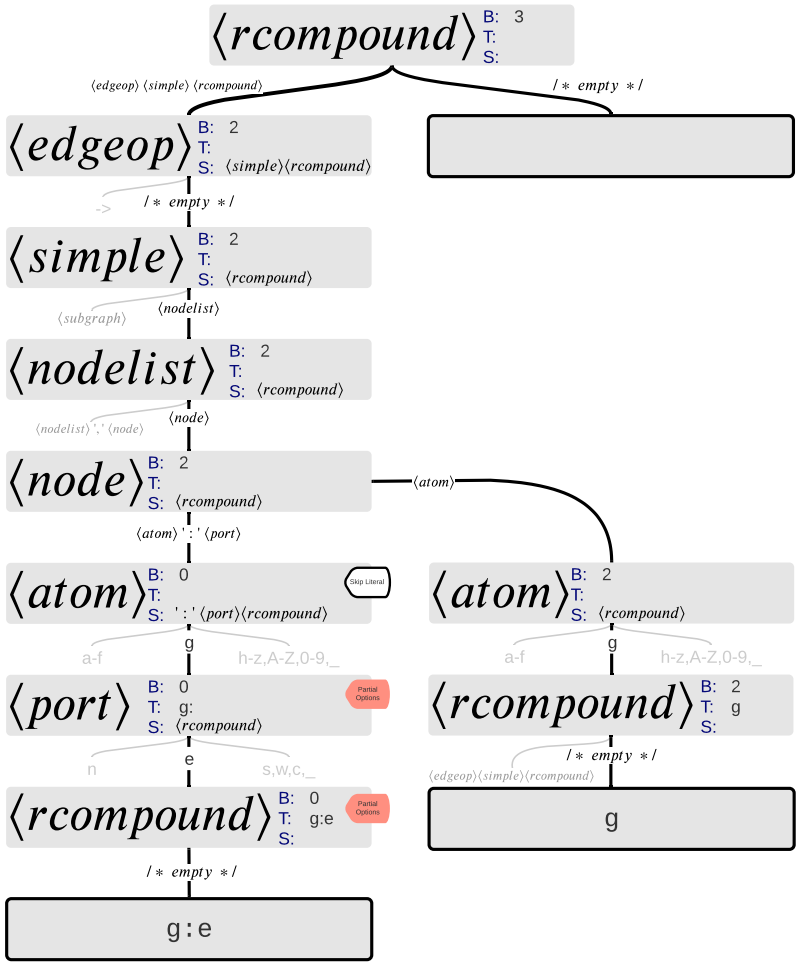


Figure 13. An excerpt of derivations based on Grammar 2 with a budget of 3. At each step, only choices consistent with the remaining budget are considered. Red tags points at which of the available steps are reduced by the available budget. White tages indicate that a literal moves from stack to text without an extra derivation step. The abbreviations B, T, and S correspond to Budget, Text, and Stack respectively.

$$\forall\ \alpha \in A\ from\ symbol(n):$$

$$\textsc{ValidOptions}(symbol(n)) =$$

$$\textsc{OptMinCost}(a) <= budget(n) + \textsc{MinCost}(symbol(n))$$

In the case of option α_1: $\langle edgeop \rangle$ $\langle simple \rangle$ $\langle rcompound \rangle$ the combined cost of the phrase is equal to 1 coming from the MinCost($\langle simple \rangle$).Thus, the budget in the produced child is reduced by 1 in the first derivation step following the MarginBudget formula given below. The formula validates if the option consumed more budget than its minimum possible cost. Then passes the appropriate remaining budget to the child.

$$\forall\ \alpha \in \textsc{ValidOptions}(symbol(n)) :$$

$$\textsc{MarginBudget}(n) = budget(n) - \big(\textsc{OptMinCost}(\alpha) - \textsc{MinCost}(symbol(n))\big)$$

The second option from the root node α_2: /* empty */[1] leads directly to terminal node. The budget in a terminal node does not matter as much as the generated text (*null* in this case).

The derivation step from $\langle node \rangle$ to $\langle atom \rangle$ ':' $\langle port \rangle$ illustrates a case where the full budget is consumed. Thus, future derivations will be bounded by their minimum cost budget. The nodes tagged in red with *partial options* in Figure 13, are given only a limited set of their options according to the possible remaining budget. In the case of the node where the symbol under evaluation is $\langle port \rangle$, the options that cost 1-byte are the only options that are allowed. Similarly, in the last derivation step from $\langle rcompound \rangle$ only the option /* empty */is allowed as it defines the evaluated symbol minimum cost.

Handling the budget is the most intricate step in tree derivation, all other variables are relatively simpler to handle. For each new valid child, all the symbols of the valid option that populated the child from current node n are added to n's stack to be the child's stack. The top of the stack is the next symbol to evaluate

[1]We use the notation /* empty */to refer to a null option.

(i.e., $symbol(child)$). If the top of the stack is a literal $l \in L$ it is translated and appended to the $text(child)$ with the budget adjust until the top of the stack is a non-terminal symbol $s \in N$. This is illustrated in the node tagged with white *Skip Literal* in Figure 13. Literals are choiceless symbols from the grammar. Therefore, we do not create explicit nodes for them to maintain decent performance.

The derivation continues until the stack is empty and there are no symbols left to evaluate. These are the terminal nodes in Figure 13. Note that it is possible for the stack to be empty but we keep evaluating the symbol in hand if the derivations lead to a single symbol each time. Thus, both the $stack(n)$ and the $symbol(n)$ must be empty to call a node n a terminal node.

Given the defined tree-based InputGenerator, it is clear how the budget does not limit the number derivation steps but it constrain it. Hence, it constrain the maximum length of the input. The derivation path that leads to a terminal with the text 'g' from the root node in Figure 13, illustrate a partial budget consumption. Therefore, the InputGenerator allows us to explore inputs of size $0 - \mathbb{B}$.

4.1.5 TreeLine Algorithm. The high-level processes of searching for expensive inputs is shown in Algorithm 1. We integrate the concept of MCTS (Section 3.2) to the InputGenerator (Section 4.1.4). Moreover, we make adjustment to the traditional MCTS algorithm such as a notion of hot-nodes buffer and rebuilding the search tree with heuristics. We describe each enhancement separately.

TreeLine algorithm expect few parameters to be defined before a search can start. However, most of these are basic parameters that are inherited either from the problem definition or the a methods we used. Moreover, we do not

Algorithm 1: TreeLine algorithm

inputs: grammar $\mathbb{G}$, number of iterations $\mathbb{N}$, budget $\mathbb{B}$, exploration threshold $\mathbb{C}$, expansion threshold $\mathbb{E}$, skip threshold $\mathbb{K}$

```
δ ← InputGenerator(𝔾, 𝔹, ℂ, 𝔼);
β ← [ ];
history ← [ ];
while within ℕ do
    node ← δ;
    with probability ∈ [0, 1] > 𝕂 do
        if HasNodes(β) then
            node ← max(UCT(β));
    node ← BestChild(node);
    feedback ← CollectFeedback(node);
    BackPropagate(node, GetReward(feedback));
    HandleHotNodes(node, β, feedback);
    TreeExhaustionEvaluation(history, δ, β, feedback);
```

hand tune these parameters for each target application. The grammar $\mathbb{G}$ is the seed we operate on. The parameter $\mathbb{N}$ is the computational budget allowed for our algorithm to do the search. The computational budget can be defined based on time or any other notion of computational constraint. We use both time and iteration as one allows us to get a sense of a wall time cost and the other empathizes the number of required target application executions. The budget $\mathbb{B}$ is what constrain the generated input length as described in Sections 4.1.3 and 4.1.4.

The parameters exploration-threshold $\mathbb{C}$, expansion-threshold $\mathbb{E}$, and skip-threshold $\mathbb{K}$ are related to how we operate the MCTS algorithm. Values of $\mathbb{C}$ are recommended to be in range [0-2] according to the MCTS literature to balance exploration and exploitation of the search tree. We try to minimize the reliance on the value of $\mathbb{C}$ by adjusting the rewards passed to the search tree (see Section 4.1.5.2). We use the value 1.5 for $\mathbb{C}$ across all experiments. The parameter $\mathbb{E}$ is also usually found in MCTS literature to balance the memory use and improve

the nodes value estimation before expanding it. Larger values of $\mathbb{E}$ means more visits to a node before it can be expanded, while smaller values means faster expansion of the tree. Memory is not an issue based on our experiments; thus, we keep $\mathbb{E}$ low ($\mathbb{E} = 20$). The last parameter $\mathbb{K}$ controls from which node we start the search for the given iteration. This is a new notion that we introduce and explain on Section 4.1.5.1. We always use the value 0.5 for $\mathbb{K}$.

In Algorithm 1 we define the notion of buffer β (line 2) and *history* (line 3). These variables are used to enhance the search result by exploring verity of potential paths faster and dropping the tree for fresh search as we have better heuristics of the target application cost range. We elaborate on these on their designated sections (4.1.5.1 and 4.1.5.3 respectively). We focus on the steps we define that align with the traditional MCTS algorithm for now.

For each pass in the search tree (Algorithm 1, line 4) we obtain a *node* to start from. We then traverse the tree to reach the best known frontier node (Algorithm 1, line 9) as shown in Algorithm 2. A leaf is a *node* that has no children yet. Thus, a leaf *node* has a broader definition than terminal nodes where terminals must have no children, empty stack and empty symbol.

Algorithm 2: BestChild function for tree traversal

```
   inputs: MCTS node
1  while not IsLeaf(node) do
2  |  children ← GetChildren(node);
3  |  node ← max(UCT(children));
4  return node
```

Following Algorithm 1 we then collect the feedback based on the frontier node we obtained (line 10). The details of collecting the feedback for a frontier

node are given in Algorithm 3. If the frontier node we get IsTerminal(*node*) we then collect the feedback from the target application (Algorithm 3, lines 1 & 2).

Algorithm 3: CollectFeedback function for random traversal or generated input execution.

```
   inputs: MCTS node
1  if IsTerminal(node) then
2  |  feedback ← execute(text(node));
3  else if IsNew(node) then
4  |  feedback ← rollout(node);
5  else
6  |  Expand(node);
7  |  node ← GetFirstChild(node);
8  |  feedback ← rollout(node);
9  return node
```

In the case where the node is new (Algorithm 3, line 3), then we apply a rollout(*node*) (Algorithm 3, line 4) from the the given node to randomly complete the the derivation steps from the given node to some terminal node. The rollout will then collect a feedback based on the given random found input. A *node* is new if it was visited less times than $\mathbb{E}$. Thus, we are not allowed to expand the node yet. If the leaf node is neither a terminal nor new, then it must be ready for expansion. The function Expand(*node*) (Algorithm 3, line 6) expands the node according to the TreePolicy defined in our InputGenerator. All new children have the same UCT value when they are newly generated ($+\infty$). Thus, we select the first child (Algorithm 3, line 7). A random rollout is performed from the new child to collect a new feedback for the given iteration.

In expanding a node we populate all children at once as given in the InputGenerator (Section 4.1.4). Thus, all valid children from the given node are added. However, there are other possible strategies to populate new children. For example, it is possible to populate children one at a time. Given the max cost

known, if the populated child is outperforming it, then it is less likely we need to populate all children and explore them all. Such strategy can be beneficial if the number of valid children is significantly large but there is small to no distinction between choosing one over the other. We choose to populate all children as we do not know how other approaches would generalize over different target applications.

The feedback of running the target application on either a rollout or a terminal node is passed to our GetReward($feedback$) function to back-propagate rewards (Algorithm 1, line 11). The reward is a smoothed value of the actual input cost. More details on the reward function in Section 4.1.5.2. The back-propagation, however, as shown in Algorithm 4 updates all nodes from the known populated node all the way to the root node. For each node in the traversed path we increase the number of visits by one and add the given reward value to the node's score.

Algorithm 4: BackPropagate function for updating the traversed path scores and visits.

inputs: MCTS *node*, reward $\mathbb{R}$

1 **while** *node is not root* **do**
2 $\quad node.visits \leftarrow node.visits + 1$;
3 $\quad node.score \leftarrow node.score + \mathbb{R}$;
4 $\quad node \leftarrow parent(node)$;

Within each iteration we evaluate the frontier node for coverage increase (Algorithm 1, line 12). Hot nodes are added to the buffer to expedite the search progress within hot-paths as we will describe in Section 4.1.5.1.

In the last step of the TreeLine algorithm (Algorithm 1, line 13), we check the search tree exhaustion to decide whether we should drop the tree and start with a fresh search tree or do more exploration with the tree in hand. The handling of feedback history and dropping the search tree is explained in details in Section 4.1.5.3.

As with similar anytime algorithms, the TreeLine algorithm runs until the computational budget is fully consumed. The larger the computational budget (regardless of its definition), the better result the algorithm would return. The inputs that maximize the cost or increase the coverage of the application are tracked during the process for post-search evaluation.

4.1.5.1 Nodes Buffer. The UCT formula balances MCTS. It uses the exploration value to help decide on which node to select, expand, or do a rollout from at each iteration. The UCT formula, given empirically known good C value, can explore the tree toward a costly input. However, given that our problem is not a win-loss type of problem, exploring a variety of high-potential paths might take longer than we aim for. As we will explain in the reward function (see Section 4.1.5.2), our rewards are in the range between 0 and 1. Traditional MCTS algorithms expect absolute binary values for rewarding. A binary rewarding module provides a straightforward notion of win or loss (good or bad in our case). Thus, it helps the MCTS algorithm have a faster convergence than our case. We introduce a node buffer of potentially good nodes to start the search from with some probability that would expedite the search toward more expensive paths.

In addition to the raw cost of an input execution, the feedback, as explained in Section 4.1.2, offers more information than just the cost. For each run we have the NewCov and NewMax indicators. These two indicators are part of the total cost. However, both NewCov and NewMax give a different perspective than the raw cost of the input effect, and they are not used to their full potential. Each time we do a rollout from some node, we only backpropagate and track the scaled cost. However, in traditional MCTS, if the rollout from the given node showed some NewCov or NewMax the information will be discarded as soon

as we finish the iteration. Tracking nodes that depict NewCov or NewMax for future exploitation could help expedite the search toward the ultimate goal of a more expensive input.

For each node that shows NewCov or NewMax we add the node to the a buffer β as shown in Algorithm 5. We only allow non-terminal nodes to be added to the buffer as terminal nodes have no further selections or expansions possible. By tracking nodes that showed NewCov or NewMax we then can start an iteration from some high-potential node instead of the root node (Algorithm 1, lines 6-8). Similar to child selection from any node, nodes in the buffer are selected based on their UCT value (Algorithm 1, line 8). Therefore, for some iterations with probability $> \mathbb{K}$ we explore a high-potential node that is either already in the good path from the root node or explore a completely different branch according to the UCT formula.

Algorithm 5: HandleHotNodes function for tracking hot nodes by adding them to the shared buffer β.

```
   inputs: MCTS node, buffer β, feedback
1  if NOT IsTerminal(node) then
2  |  if NewCov(feedback) ∨ NewMax(feedback) ∨ NewCost(feedback)
   |   then
3  |  |  β ← β ∪ {node}
```

To illustrate the behavior of the buffer according to the nodes' UCT value, we conducted an isolated experiment. In this experiment, we track the UCT value for each node added to the buffer from the moment it is tracked until the end of the experiment. The goal is to understand how these nodes' UCT values increase over time. Hence, they are likely to be selected from the buffer as a starting node for an iteration. Figure 14, illustrate the complete view of the UCT value progress for each added node. The x-axis in the figure represents iterations, and the y-axis

represents the UCT value for each node. Each line in the plot represents a unique node. Line colors are random but help distinguish a group of nodes added earlier (blue) in the experiment versus ones added later (black) and the ones in between. This high-level view shows that a large number of nodes is added to the buffer. However, it does not help us understand the UCT value progress.

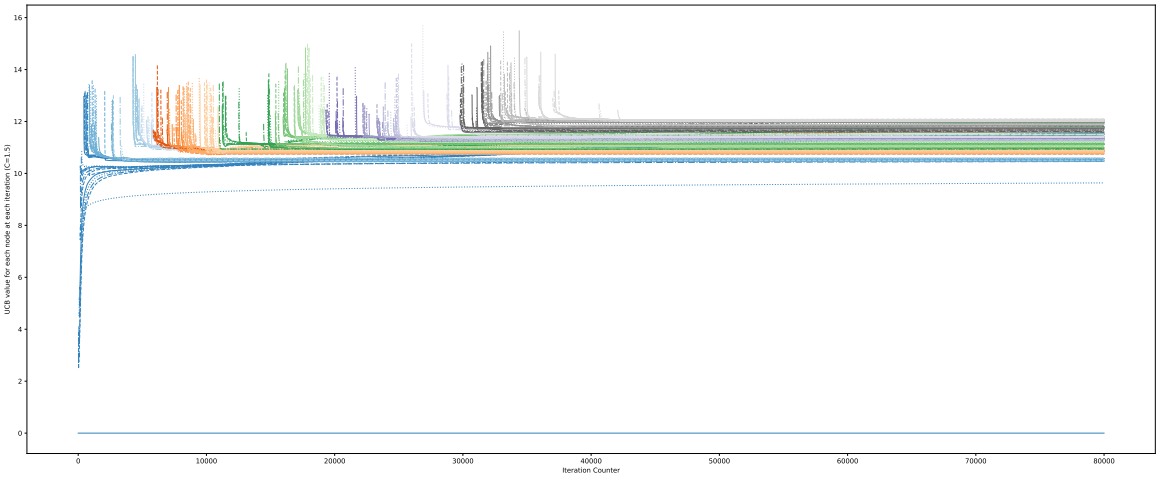


Figure 14. The buffered nodes' UCT value change over iteration of an isolated experiment using `libxml2`. The x-axis represents iterations, and the y-axis represents the UCT value for each hot node node.

The illustration in Figure 15 shows a zoomed-in version of the same experiment on the y-axis (values between 10 and 12.25). This view shows how the UCT value of a sub-set of nodes are increasing over iterations. It indicates that not all nodes in the buffer have high potential. Furthermore, the figure shows how fast the UCT value grows for some nodes compared to the others. It is hard for each iteration to know what subset of the shown nodes falls within the known expensive path at that iteration. However, it is safe to assume that some node with max UCT value in the buffer is not in the hot-path from the root node for some iterations. Thus, selecting it would speed up the exploration of its path compared to the hot path from the root node.

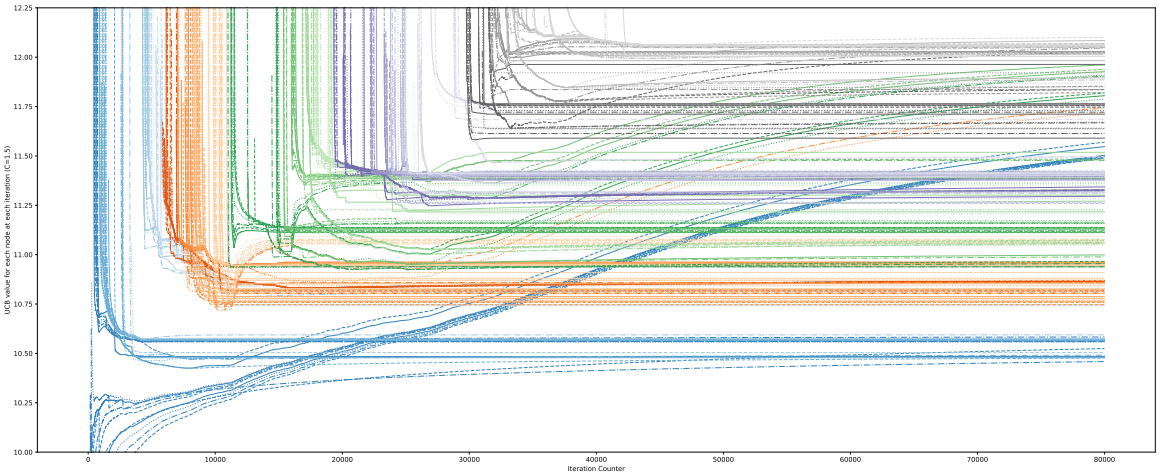


Figure 15. A zoomed-in version of the buffered nodes' UCT value change over iteration of an isolated experiment using `libxml2`. The x-axis represents iterations, and the y-axis represents the UCT value for each hot node node.

In addition to expediting the search process by affirming an exemplary node or rolling it out, the buffer β helps us avoid falling into local maxima. Within the first few iterations the search tends to be a breadth-first search. As more information is gathered from executions, MCTS tends to exploit a few selected branches relative to the root node. Keeping track of nodes that has NewCov or NewMax and exploiting them allows us to balance the exploration of high-potential paths other than the ones that fall within the UCT formula's reach.

The nodes added to the buffer β can be nodes at a very high level of the search tree. Thus, a node that showed some NewCov or NewMax based on applying a random rollout might be less expressive with how to regenerate the same or closely related input. Hence, it is less likely to be helpful in exploiting the particular increase or maximization in coverage of an edge in the control-flow graph. Mitigation can be applied by changing how rollouts works by and populating all the nodes from the root to a terminal where the input showed NewCov or NewMax. However, the exact path where NewCov or NewMax

is observed is not what we aim for. It is the potentially slight variations from the given path that might have some potential. Thus, although nodes at a high level in the search tree might be a broad representation of the NewCov or NewMax case, they are still a good starting point for selection, expansion, or rollout if they have a good UCT value compared to other nodes in the buffer β.

4.1.5.2 Dynamically Adjusting Rewards. Each target application has its unique cost range. Based on the size of the control-flow graph, the provided grammar, and the allowed budget, the possible range will change. In our experience, the ranges are radically diverse. Some will have ranges as low as in thousands, and others will be in billions. Moreover, naive inputs such as an empty input would not cost 0. The simplest possible inputs would have some cost larger than 0. The range between the minimum cost and the maximum known one is an application-dependent value. These factors enforce the use of a reward function that adjusts the feedback to some known range.

Moreover, the cost of executing an input is not always good enough to drive the search toward more expensive inputs. The length of the input is an important factor in finding the most expensive input. For example, the input `1 2 3` for an application such as QuickSort will depict the worst-case of a sorted array. However, if the budget allowed is 10, even though the input shown of length 3 captures the pattern of the worst-case input, it is not the most expensive possible input given the allowed budget. Therefore, spending time on paths that consume a lower budget than allowed is, in most cases, a wasted computation effort.

A counter-argument to the push toward the total consumption of the budget is the possibility of depicting a worst-case execution with low budgets (e.g., an infinite while loop in a programming language can be written in 10 bytes).

However, input generation profilers are usually used to find some pattern rather than obvious exceptional cases. Moreover, shorter cases can be included in more extended input. Therefore, biasing toward longer inputs encompasses both.

We scale the reward for execution cost between 0 and 1 with t-digest, a streaming quantile estimation function (Dunning, 2021; Dunning & Ertl, 2019). Quantiles in statistics are cutting points for a range of data. And a streaming estimation of quantiles is the adjustment of the defined cutting points in a continues data observation environment. Hence, it is a good fit for our problem as we do not know the range of cost a priori. With t-digest, regardless of the application cost range, we will always get a reward between 0 and 1. Moreover, the quantile function helps distinguish more expensive costs as they are discovered and adjust dynamically. As the t-digest observe more raw cost is becomes more accurate in its estimation. Similarly, we adjust the generated input length using a linear function to fit the known length minimum and maximum values between 0 and 1.

The reward function returns a balanced weighted value in the range $\Theta = [0.1, 0.9]$ of the smoothed cost and length values. Initially, the reward function starts by assigning the most negligible weight to the length (0.1). Then according to observations, the weight is adjusted slowly in both directions.

$$\textsc{GetReward}(feedback) = \textsc{Quantile}(cost) * (1 - \Theta) + \textsc{Scale}(len(input)) * \Theta$$

In a case where the grammar and the target application often lead to longer inputs, the weight will be in its lowest case. Otherwise, the weight will be increased to a point where the actual length value is used. The weight adjustment in the reward occurs only within the first established search tree (Section 4.1.5.3).

4.1.5.3 Search Refresh.

The nature of the problem and how we designed our reward function (Section 4.1.5.2) implies that the same inputs can be rewarded differently over time. Because the search space is unknown, the rewards at best are a good approximation of the cost space. As the number of target app execution on expensive input grows, the estimation becomes better. For example, assume we generate the input i_1 at time t_0, costing 10K as the most expensive input observed at time t_0. Then assume we generate input i_2 at time t_{10} where the cost of i_2 is 20K as the new most expensive input. In this case, the reward function will be adjusted according to the cost of input i_2. Thus, any future observation of input i_1 after time t_{10}, will not get the same reward as given at time t_0. The reward given to any non-maximum input will be relatively larger than the reward given to the same input at any future time.

Even with a smoothed reward function, having a better approximation of the cost space of a target application causes the search to be skewed toward initially observed expensive input. Unfortunately, tracking rewards within each node for future reward value correction as we observe new expensive inputs is computationally infeasible. Thus, our next best solution is to thoroughly explore the current search tree given the anomalies in its reward accumulation before we refresh the search tree and start over. The intuition is that the current accumulated rewards within the search tree at worst must have led the search to some local-maxima. Because reaching a good state of the search tree is expensive, we aim for making sure we exhaustively search the established space before dropping the tree and starting over (Algorithm 1, line 13).

The details of evaluating the tree search exhaustion are given in Algorithm 6. We first update the history of observation since the establishment

of the tree in hand. Next, if we satisfy the number of required observations (tail) and stability definition, we drop the tree by clearing all the tree-related variables and establishing a new tree root. It is important to note that the observed reward estimate and the coverage information from the target application are carried across trees.

Algorithm 6: TreeExhaustionEvaluation function for evaluating and possibly refreshing the exploration progress of an established tree.

inputs: the execution cost history $history$, the tree root δ, buffer β, the most recent execution $feedback$

1 $history \leftarrow history \cup \{feedback\}$;
2 **if** len$(history) > tail \wedge$ HasStabilized$(history[-tail:])$ **then**
3 | Clear(β);
4 | Clear$(history)$;
5 | Drop(δ);
6 | $\delta \leftarrow$ InputGenerator$(\mathbb{G}, \mathbb{B}, \mathbb{C}, \mathbb{E})$;

The function HasStabilized measure the probability of search effectiveness given the current tree (Algorithm 7). It looks into the last *tail* number of observed costs from *history*. Given the passed set of most recent observations, HasStabilized measure the percentage of redundant costs observed compared to the set provided. A more diverse set of costs indicates exploring a wide range of inputs and paths given the current search tree. On the other hand, a less diverse set of costs indicates we are exploring similar inputs and devoting more computational effort to the same tree is probably a computational waste.

A problem with our previous definition of stabilization is that the cost range for target application is widely diverse as described in Section 4.1.5.2. Target applications with a smaller cost range will tend to have less diverse input regardless of the search tree state. Therefore, we define a dynamic cutting threshold of what percentage of redundant costs is allowed to determine a search tree

Algorithm 7: HasStabilized function for stabilization evaluation.

```
   inputs: a sub-set of observations history ℍ
1  unique_percentage ← len(NumUniqueValues(ℍ))/len(ℍ);
2  if unique_percentage < ϵ-refresh(Q) then
3  |  return true
4  else
5  |  return false
```

HasStabilized. Moreover, we determine the *tail* size according to the known range. Applications with more extensive ranges will use smaller *tail* sizes and vise versa.

The primary concern in defining the tree refresh strategy is to waste as least computational effort as possible. Thus, we define our refresh threshold based on the computational effort spent. We keep track of the number of target application executions processed (Q) for each established search tree. Using the $\epsilon - refresh(Q)$ formula below, we can control how the cutting refresh threshold grows given the number of executions made.

$$\epsilon - refresh(Q) =$$
$$1 - \left(rate_{min} + (rate_{max} - rate_{min}) * \mathrm{e}^{(-1 \, * \, Q \, * \, rate_{decay})}\right)$$

The variables $rate_{min}$, $rate_{max}$, and $rate_{decay}$ are user-defined and fixed values. We use the value 0.1, 1, and 0.00005 respectively for each variable. Given the ϵ-$refresh(Q)$ function, the threshold will then start from 0.0 and grows slowly with each execution up to 0.9. The intuition behind the dynamic cutting threshold is that as each search tree is established, it is less likely to exhaust the search even if we have many redundant costs observed. However, as the number of executions

grows, the established tree must have good exploration distribution. Therefore, with each new execution, the threshold is less tolerant of redundant observation.

Similar to the cutting threshold, the *tail* size is adjusted based on the observed range. We keep this simple by defining bucketed ranges between 2500 and $200,000$. Given the size of the cost range, we adjust the *tail* size. More extensive ranges are expected to have more diverse costs; therefore, we assign them smaller *tail* sizes and vise versa.

Each time we start the search with a new search tree, we reward input exploration based on the most accurate reward estimate. The search from a new search tree is either going to lead us to the same expensive input in the case where we found the *max* or will be offered the opportunity to find a new expensive path based on the most recent reward function much faster.

4.1.5.4 Avoiding Search in Exhausted Subtrees.

In finding the most possibly expensive input, the goal is not to learn the optimal path but find it. Thus, it is ideal for populated nodes in the search tree if we traversed them a minimal number of times. It is safe to say a node is traversed enough time if it is a terminal node as it has no possible children. Thus, we define a node locking strategy to minimize the number of visits to an already explored path.

Locking a node simply changes its UCT value to be negatively infinite ($-\infty$). Therefore, a locked node will only be selected if all other nodes are locked as they have the same UCT value.

We can only determine to lock a node if it is populated (not part of a random rollout). Moreover, we can only lock terminal nodes or one that has all its children locked. Therefore, the locking strategy works from a bottom-up approach on populated nodes. Hence, it will only be effective at a late stage of the search

process. Nevertheless, it ensures that as soon as we determine the cost of some path, we never revisit the same path, even if it is the most expensive path.

The locking strategy, the UCT formula, and nodes buffer provide a method to explore as much unique inputs as possible. However, contrary to the UCT formula and node buffer, the locking strategy applies a backward refinement to the tree rather than a forward exploration. It is essential to highlight that the locking strategy is less likely to be effective with search tasks of large budgets. The computational effort required to populate deep paths (reaching a terminal node) are usually significantly large. Thus, in most cases, locking acts as a safety check to not waste computational effort rather than an enchantment.

4.1.5.5 Biased Rollouts. A rollout in MCTS can complete a sequence of moves completely randomly or with some heuristic bias. Grammars tend to have a large set of options in some production rules. Most commonly, we would have options similar to the ones given on $\langle atom \rangle$ from Grammar 2. The consequence on cost of selecting `a` versus `A` is an application dependent. However, often such options represent a variable naming. Thus, it is more important to distinguish a pattern based on past selections rather than randomly making a decision.

Selecting the best possible decision at each choice requires keeping track of all potentially relevant factors (including the full path to that choice) in which it has been selected before. Unfortunately, tracking full context for each possible decision is computationally infeasible. Moreover, populated nodes based on the MCTS algorithm already represent the full context of observed derivation steps. A simple solution that tracks *good* choices based on partial context can reduce the exploration necessity of grammars with a large set of options.

We devise a bigram biasing selection based on rewarded and penalized grammar options. For each non-terminal symbol $s \in N$, the bias will track weight for each option given all the non-terminal symbols that might lead to s. Each time a sequence of bigrams lead to an input with NewCov, NewMax, or NewCost the weight is increased significantly. Otherwise, the weight is gradually decreased for each idle cost observation of pair symbols.

The bias weight of bigrams is only used in rollouts. Hence, the algorithm follows the UCT formula for populated nodes. However, within each rollout, we perform educated random selection to some terminal node.

4.1.6 Notable Implementation Details. Here we provide a few additional details about our current implementation of TreeLine.

Docker container. To further expedite building, testing, and extending TreeLine, we have implemented it in a Docker container with all the legacy dependencies of AFL. We split our prototype into two different processes. One running AFL, and the other runs our core implementation of the search process. AFL handles the target application instrumentation and runs. All other processes are handled by TreeLine. Using Docker provides some flexibility on distribution and replication.

We have implemented the main logic of TreeLine as a Python program communicating over a Unix socket with a version of AFL that builds upon the extensions earlier built by the developers of PerfFuzz. This architecture has a performance cost, not only because the Python interpreter is much slower than compiled C code in AFL and because AFL is idle between requests, but also because each test execution requires inter-process communication to send a test input to AFL side and another to receive feedback. This could be ameliorated by

building a single process executable with Python's embedding support, but we did not think this is a significant limitation to adopting such a solution. Using Python was a decision taken to expedite experimentation and ease of change compared to C.

The rollout and the target application run are the most computationally expensive task in our current process. To put this into perspective, currently, we are processing at best 80 inputs per second while PerfFuzz is capable of processing $1,508$ (18.85x) inputs per second for the same benchmark (`libxml2`).

Timeouts. Although we search for slow test cases, sometimes TreeLine finds test input that is *too* expensive. We set a timeout of ten seconds on test case execution. Allowing unbounded runs could lead to wasting the entire computational budget over very few generated inputs.

Lazy search. We do not exhaustively search for the reachable node with maximum UCT value each time we draw from the buffer of hot nodes. Since the UCT values change slowly on most iterations, we look only at a smaller buffer of nodes with the top 10 UCT values. Periodically (every 500 iterations), this buffer is updated.

4.2 Evaluation

We experimentally evaluate TreeLine with respect to three research questions.

RQ1. How does the performance of the TreeLine approach compare to mutational fuzzing with limited expenditure of computational effort?

RQ2. Do heavy rollouts with simple bigram bias improve search effectiveness in TreeLine, relative to light (uniform random) rollouts?

Benchmark	Description	Version	SLoC
`wf`	Simple Word Frequency Counter	0.41	394
`libxml2`	The XML C parser and toolkit of Gnome	2.9.7	191,371
`graphviz`	Graph visualization software	2.47.0	1,050,696
`flex`	Generator of lexical analyzers	2.6.4	22,260

Table 1. List of benchmarks used for evaluation and their properties.

RQ3. How does the TreeLine approach scale with an increase in the length bound for generated inputs?

The first RQ relates MCTS-based input generation to an alternative mutational approach. The latter two are questions about effectiveness of improvements on random rollout and rewarding.

We conduct our experiments on a Docker container hosted on a modest workstation (2017 iMac with 3.8Ghz i5 and 8GB memory). Aside from stopping other user applications for consistency, the workstation was running in its normal state (e.g., it was not isolated from the department network). The Docker container was given 8GB of memory and access to all four CPU cores.

Although AFL (Zalewski, 2013) allows the use of plugins to instrument programs in any language, we use its default settings to work with C/C++ applications. We conduct experiment using four real-world C/C++ applications. We choose two that were previously used to evaluate PerfFuzz (Lemieux et al., 2018): `wf` (de Barros, 2021), and `libxml2` (Veillard, 2017). We add two new benchmarks `graphviz` (AT&T Labs Research, 2021) and `flex` (Paxson, 2017) that are widely used software tools with available source code, and which exemplify the class of application, driven by richly structured input, that TreeLine is intended to address. The details of all the benchmarks are given in Table 1.

The selected benchmarks span a variety of sizes. We count SLoC using the utility `cloc` to count source line of code for the core language in each benchmark (c/c++) and skipping appropriately marked test files.

The `wf` application requires no particular input structure. While `libxml2` processes structured text, it is only a parser whose behavior is not driven by that structure beyond accepting or rejecting it. In fact fully valid XML is a cheap input for `libxml2`; its taxing inputs are invalid files. In contrast, `graphviz` processing is controlled by inputs that meet the syntactic requirements of its *dot* input language, although that syntax is minimal and forgiving. Mutational fuzzing tools still have a good chance of generating valid inputs for `graphviz`. `flex` expects valid regular expressions and has a richer input language than `graphviz`. Like `graphviz`, the processing of `flex` is controlled by a valid input file, and (as far as we know) rejecting syntactically incorrect files is not among its worst case behaviors.

Prior published evaluations of PerfFuzz used trivial seeds consisting of strings of zero bytes. Such seeds would disadvantage PerfFuzz in a comparison with TreeLine, so we instead collect seed inputs from official documentation as available. As required by AFL, we adjust the size of seed inputs to be within maximum allowed input budget, either breaking them into multiple valid seeds or removing redundant keywords. In the case of `flex` we omit C code from inputs to keep input reasonably sized and additionally provided PerfFuzz with random but valid seeds produced by TreeLine. We manually composed seed inputs that capture all the possible grammatical patterns in the grammar for TreeLine.

For TreeLine, we construct grammars for each benchmark based on official documentation or an existing parser. We transliterated the yacc parser from `graphviz` to the form we required by trimming extraneous code. We created a `flex`

grammar based on a page of documentation. In the case of `wf` and `libxml2`, which expect less structured input, we constructed very simple grammars by hand. For example, for `wf` we construct a trivial grammar of character sequences composed from the English alphabet and whitespace. For `libxml2` we construct a grammar that includes XML special characters (e.g., opening `<` or closing `>` tags.) and the English alphabet for tags or content. All constructed grammars are given in the Appendix.

Experiments can be conducted based on different computational budgets. For example, we can run TreeLine for T hours or specify the number of target-application execution allowed. We primarily used a time budget of 60 minutes, but also conducted some experiments with a limit of 200k executions to evaluate (by comparison with time-bounded experiments) the impact of generation speed.

The efficiency of grammar-based and mutational fuzzing are impacted by the length bound on generated inputs. Longer inputs take longer to generate, but may be necessary for applications that take richly structured input. A length bound as little as 10 could be enough to find performance issues in `wf`, but for applications like `graphviz` with keywords like "digraph" and recursively nested structures, longer inputs may be required. We use a budget of 60 bytes as a base across all benchmarks comparing TreeLine to PerfFuzz, and longer inputs for exploring RQ3.

For each benchmark we keep track of NewCost to measure tool's effectiveness in finding new expensive inputs. We also track the maximum hotspot to measure the tool's effectiveness in finding the program component that maximizes the cost the most. A hotspot is simply the edge with the maximum hits given some input. We keep the maximum hotspot observed over all generated

Benchmark	**TreeLine** exec/s	**PerfFuzz** exec/s
`wf`	61.25	2936.80
`libxml2`	79.32	1507.81
`graphviz`	46.20	729.45
`flex`	1.33	162.78

Table 2. Average number of target application executions per second (exec/s) for each benchmark across 20 runs for each. Mutational fuzzing in PerfFuzz is much faster.

inputs. Moreover, we track the generated input size as it shows the search progress concentration in term of budget consumption.

4.2.1 How TreeLine Compare to Fuzzing-Based Techniques.

Ideally we would compare an implementation of TreeLine to a variant implementation that is identical in every regard except that it uses mutational text fuzzing and a genetic search (the AFL approach) rather than MCTS search with grammar-based generation. PerfFuzz is not quite that ideal comparison, but it is the state-of-the-art tool for finding pathological inputs, and it is built atop the same AFL engine for execution monitoring, with some extensions that we have also used.

A clear advantage of mutational fuzzing over tree-based search is the speed with which inputs can be generated. AFL is particularly well-engineered to generate and test a vast number of mutants very quickly. The difference in speed of execution between TreeLine and PerfFuzz ranges from 15.8x times slower for `graphviz` to 122.5x slower for `flex`, as shown in Table 2. Clearly TreeLine must generate *much* better inputs on average to compete.

The impact of raw generation speed is illustrated by comparing the progress of TreeLine and PerfFuzz in Figure 17 to their relative progress in Figure 16.

It is clear that the primary advantage of PerfFuzz for these benchmarks is in producing test cases much more quickly.

From Figure 16, in applications that require little structure PerfFuzz is likely to eventually find more expensive inputs than TreeLine. The `wf` is a micro application with no expectation of any input structure. From the 20 experiments conducted, PerfFuzz reach a very high confidence level (minimal blue shaded area in Figure 16a). A sample expensive input generated by PerfFuzz shown below[2].

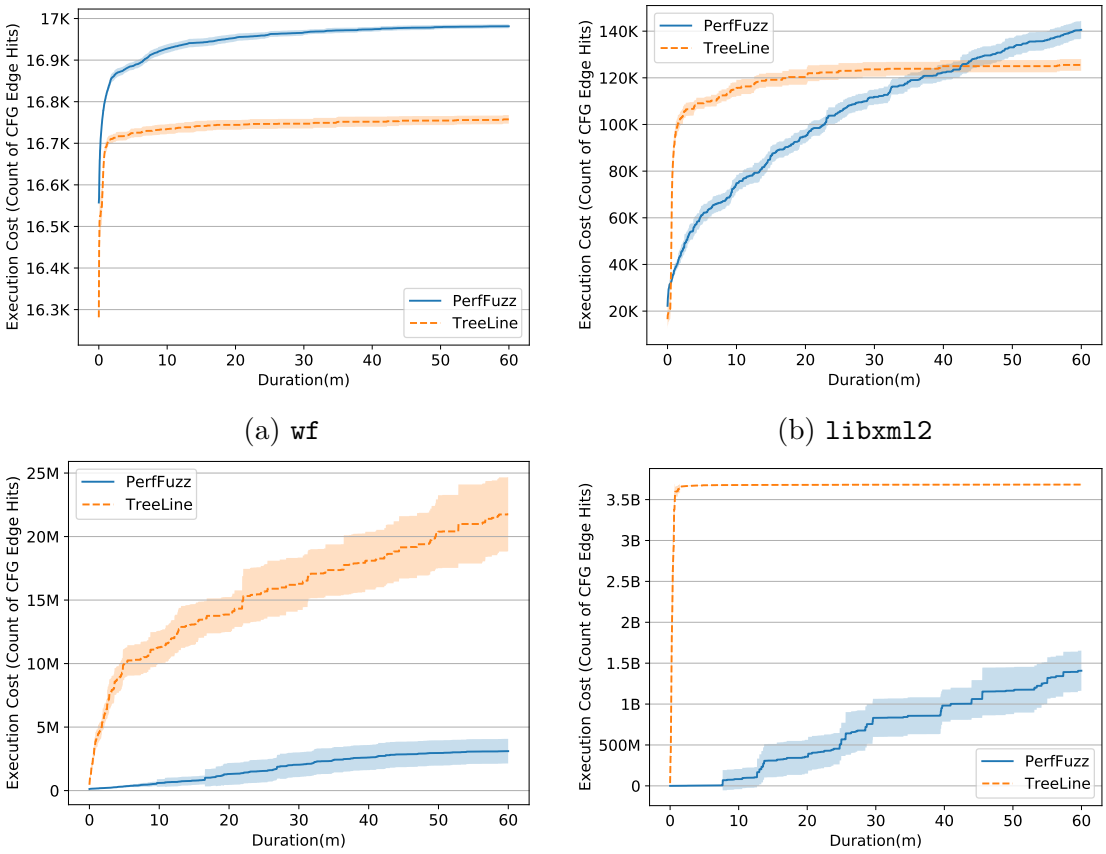
```
x\n ï\n l\nM\nw\n u\n f\n b\n å\n Ï\nD\n i\nT\n a\n t\n d\nO\nSS\n v\n e\n y\n s↩
    ↪\nG\n ó\n ñ\n j\n q\n r\n n\n z
```

The main pattern captured hints that maximizing the number of words by placing the shortest possible words into a dedicated line for each leads to a more expensive input. Following the same sampling from TreeLine we obtain the input below.

```
T d2S\nO4Z\n\nY\n c6\n u\n\nDb\n q a 02y4X\n\n k\n g\n n\nK6XP \n\n↩
    ↪A6V3\nH9y\n D
```

The input from TreeLine has the same general pattern, but PerfFuzz refines it more effectively. Note that while TreeLine uses the full English alphabet, PerfFuzz generates a much wider family of characters by random generation. The PerfFuzz developers report effectiveness at finding hash collisions (Lemieux et al., 2018), which the more limited alphabet used in our grammar is less likely to generate. The average maximum cost of input found by PerfFuzz is 1.01x as expensive as the average maximum cost found by TreeLine. For this class of application, mutational fuzzing is at least as effective as TreeLine, although the advantage is not as large as we might have expected.

[2]The symbols ↩ and ↪ indicate we broke the line to fit the input within space. All other colored sequences represent unprintable characters, such as escape sequences.



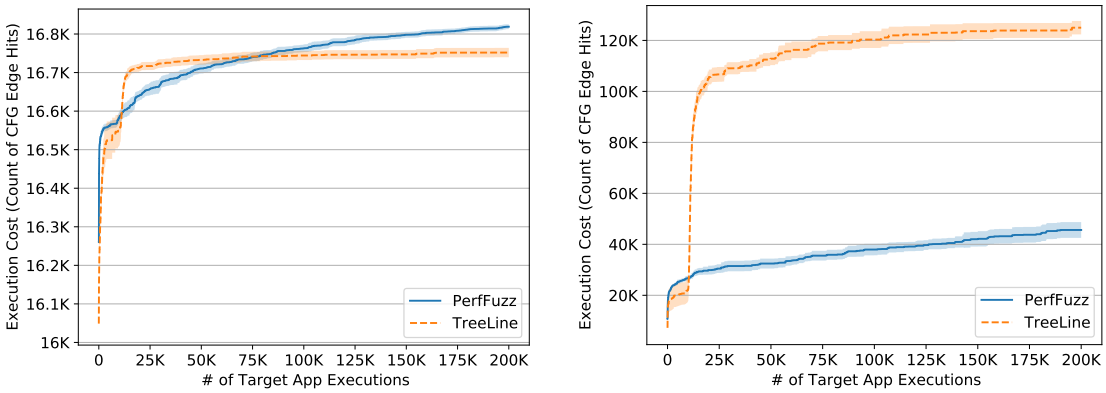
(a) `wf`

(b) `libxml2`

(c) `graphviz`

(d) `flex`

Figure 16. Time-based comparison between TreeLine and PerfFuzz showing maximum path length found throughout the duration of the 1-hour (scaled by minutes). Lines and bands show averages and 95% confidence intervals across 20 repetitions. PerfFuzz eventually finds better inputs for `wf` and `libxml2`. For `graphviz` and `flex`, TreeLine dominates and finds more costly inputs in 10 minutes than PerfFuzz finds in an hour.



(a) `wf` (b) `libxml2`

Figure 17. Execution-based comparison between TreeLine and PerfFuzz on `wf` and `libxml2` showing maximum path length found throughout 200K executions. Lines and bands show averages and 95% confidence intervals across 20 repetitions.

On the other hand, applications that require structured input (Figures 16c & 16d) to test the core functionality impose a challenge for PerfFuzz. For `graphviz` and `flex` the average maximum cost found by TreeLine are respectively 7.46x and 2.73x as expensive as the average maximum cost found by PerfFuzz. A representative maximum input from TreeLine in the case of `graphviz` (shown below) created a complex cyclic digraph that embeds a complete bipartite digraph on a subset of its nodes (Figure 18).

```
digraph{o,b,p,9,N,g,_,1,7,3,T->g,d,r,9,v,j,5,w,1,_,7,v,UUG_}
```

To maximize the input cost, it is not enough to create a bipartite digraph; a balance between the bipartite part and cycles must be found. For example, even though they represent a bipartite digraph the hand crafted inputs below, a bipartite graph and a graph in which all nodes participate in cycles respectively, are 10.37x to 26.45x cheaper than the input found by TreeLine.

```
digraph{a,b,c,d,e,f,g,h,i,g,k->l,m,n,o,p,q,r,s,t,u,v,w,UUG_}
```

```
digraph{o,o,o,o,o,o,o,o,o,o,o->o,o,o,o,o,o,o,o,o,o,o,o,UUG_}
```

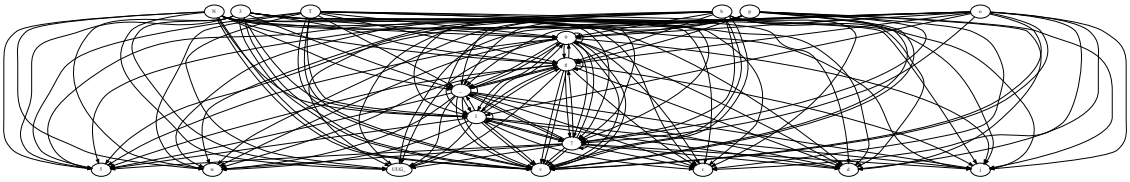


Figure 18. A `graphviz` rendering of the most expensive input found by TreeLine. The input created a complex cyclic digraph that embeds a complete bipartite digraph on a subset of its nodes

A special case is shown in the case of flex (Figure 16d). TreeLine found an input so expensive it triggered a 10-second timeout in less than 2 minutes across all 20 experiments. A selected random maximum input from TreeLine triggers the warning for the known "`dangerous trailing context`" (Paxson, 2001) from `flex`'s documentation.

```
%%\n.{31}{13}{0,65}/"\]"(\]+) ;\n%%\n
```

To maintain consistency for this evaluation we do not make any changes to the grammar or timeout values. However, an advantage of a grammar-based input generator is the ability to steer the grammar off generating such input. If the performance bug is known by the developer and accepted as it is, then a grammar can be tailored to focus on other or more specific modules.

Another perspective we can use to measure the effectiveness of each approach is to look at the single application component the maximize the cost. In other words, we look at the application's hotspot, which is the single edge in the control-flow graph with the greatest hits. Figure 19 shows our findings again compared to PerfFuzz. In all cases, the maximum hotspot is consistent with each tool's overall finding in terms of cost (Figure 16). However, an interesting observation is shown in the case of `wf` (Figure 19a). Although PerfFuzz finds inputs that maximize the overall cost, both TreeLine and PerfFuzz find

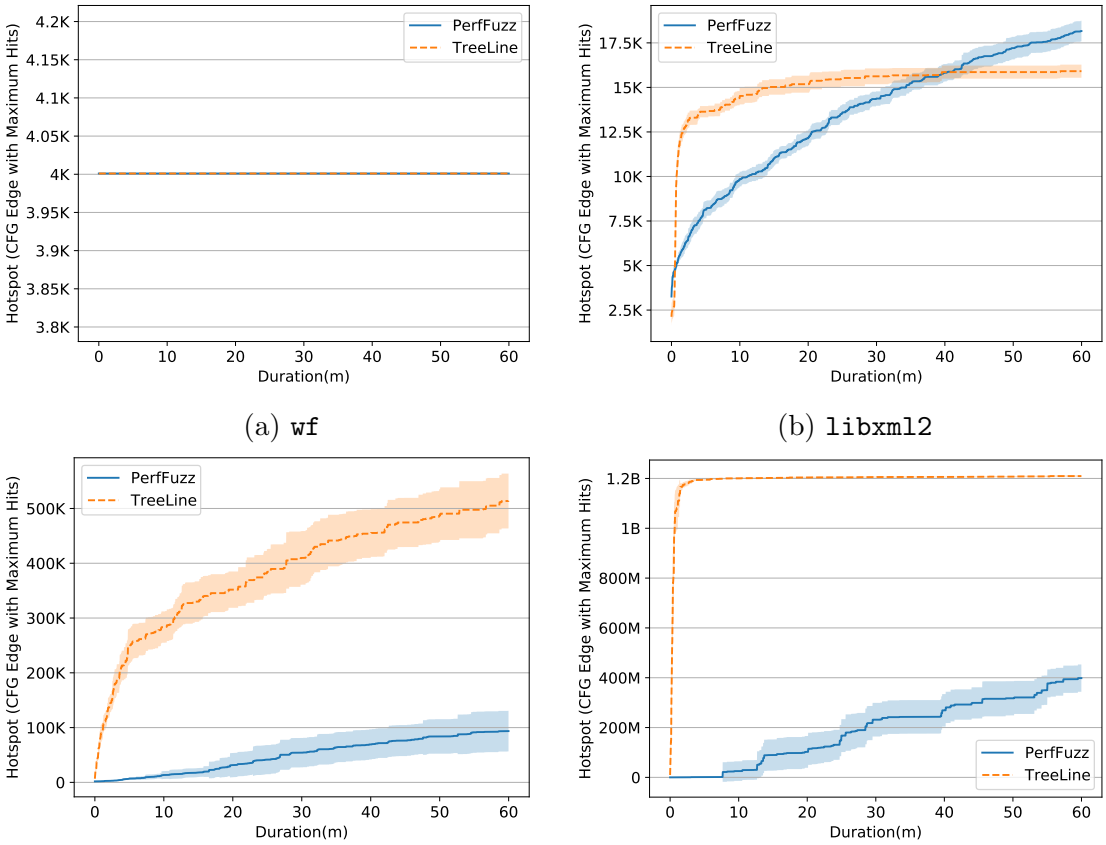
the application's hotspot immediately. The similarity in hotspot value for `wf` is consistent with our conjecture that a general pattern is found by TreeLine.

4.2.2 Biasing Choice for Heavy Rollouts. In grammars where there is a large set of equally important options if broken from their context, we devised the use of heavy rollouts (bias) to maintain a partial-context-based when exploring the tree. Results presented in Section 4.2.1 are all use heavy rollouts with biased choice of bigrams. The bigram bias table has a cost. Thus, we ask how effective is keeping a partial context for all the benchmarks on overall performance? Figure 20 shows the progress of TreeLine with and without bias for each benchmark.

A clear advantage of using heavy rollouts is illustrated in the case of `graphviz` and `flex` (Figures 20c & 20d). Although less observable in the case of `flex`, the bias allows TreeLine to reach a more expensive input in a shorter time. In the case of `graphviz` the maximum cost found using heavy rollouts is 1.58x as expensive as random rollouts.

The same pattern does not hold in the case of `wf` and `libxml2`. Also, the maximum cost found for `flex` is not significantly different with heavy rollouts. However, in line with intuition, the confidence is higher when using heavy rollouts with bigram bias than with random rollouts. This suggests there will be more consistent results using heavy rollouts than random rollouts.

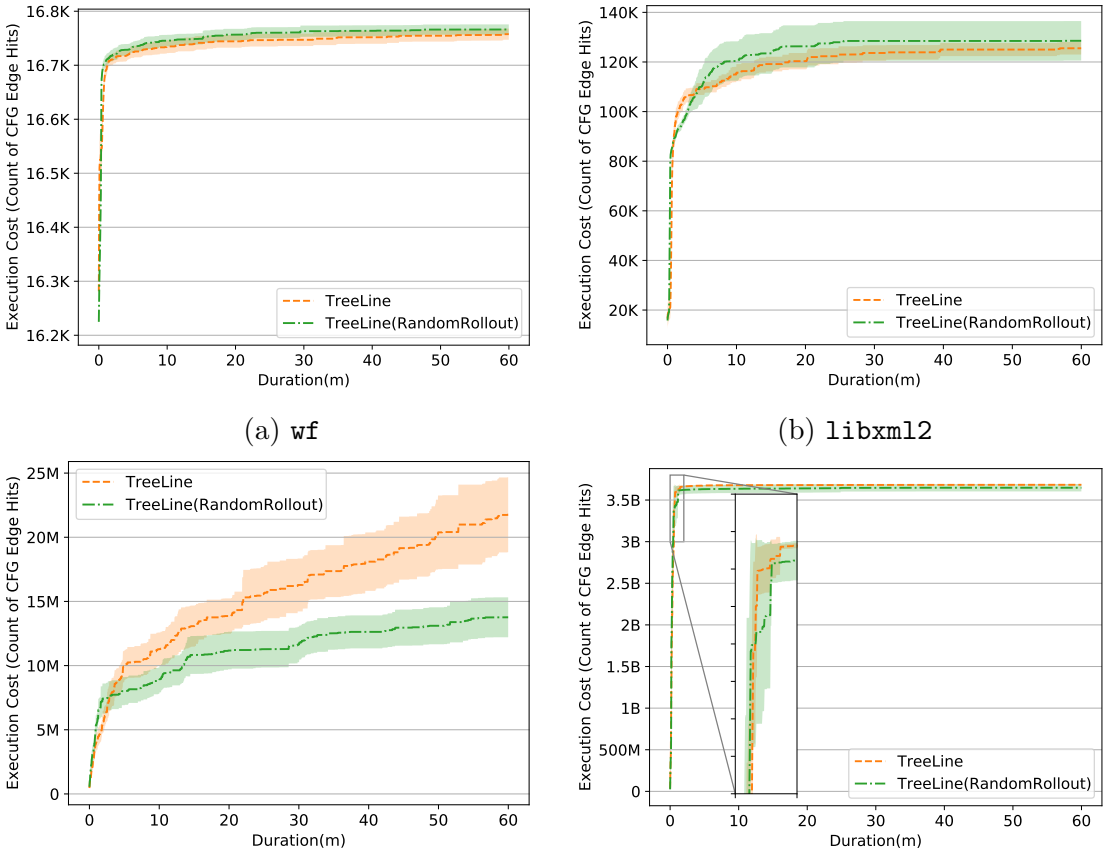
4.2.3 TreeLine Scalability for Generating Longer Inputs. The length limit for generated texts impacts performance in both mutation-based and grammar-based generators, so they are often limited to creating short test cases. However, longer inputs may be needed for exploring rich syntax. The experiments described above use the same 60-byte limit as prior published evaluation of



(a) `wf` (b) `libxml2`

(c) `graphviz` (d) `flex`

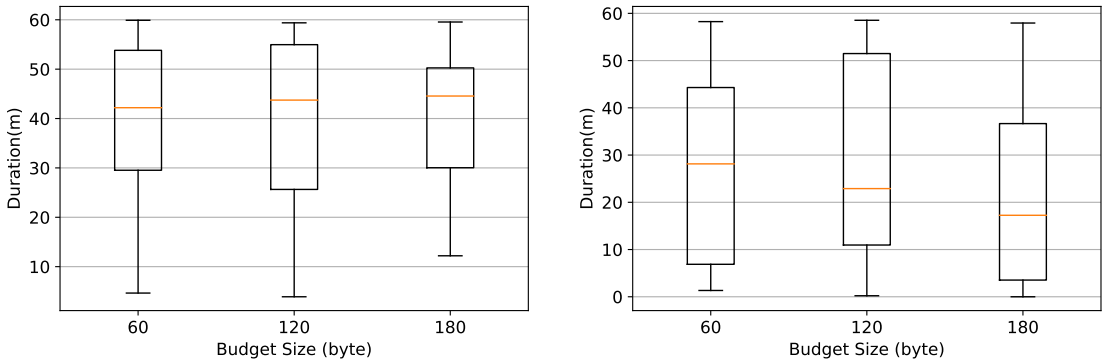
Figure 19. Time-based comparison between TreeLine and PerfFuzz showing maximum hotspot found throughout the duration of the 1-hour (scaled by minutes). Lines and bands show averages and 95% confidence intervals across 20 repetitions. The results are consistent with the overall performance cost. However, for `wf` TreeLine actually finds the exact hotspot found by PerfFuzz even if the overall cost found by PerfFuzz is higher.



(a) `wf` (b) `libxml2`

(c) `graphviz` (d) `flex`

Figure 20. Time based comparison between TreeLine with and without bias rollouts showing maximum path length found throughout the duration of the 1-hour (scaled by minutes). Lines and bands show averages and 95% confidence intervals across 20 repetitions.



(a) `graphviz` (b) `flex`

Figure 21. Summary of the time it takes TreeLine to find the cost-maximizing input for each run. Each budget was repeated 20 times.

PerfFuzz and SlowFuzz. If we double or triple that limit, to 120 or 180 bytes will TreeLine still be as effective?

If TreeLine scales well to these longer bounds on generated input size, we expect to see three things: TreeLine should still find its worst case input reliably within an hour, TreeLine should make full use of the allowed length (rather than getting stuck exploring shorter texts), and the patterns found with shorter bounds should be found also with the longer bounds (it doesn't get lost), although we would also accept new and better patterns. We only consider `graphviz` and `flex` for these experiments.

Speed of finding maximum cost input. We consider the time distribution required for TreeLine to settle on each max cost (Figure 21). In the case of `graphviz` (Figure 21a), a few searches continue finding new maximum costs until the last possible second. For the budgets 60 and 120, TreeLine occasionally found the maximum cost input within the first 10 minutes. Typically, however, an input that maximizes cost is found within 40 to 50 minutes.

budget	cost	flex	graphviz
60	min	3,673,195,978	10,244,177
	max	3,696,182,810	33,831,034
	avg	**3,683,384,935**	**21,763,449**
120	min	3,235,375,950	92,440,322
	max	3,694,925,552	444,512,672
	avg	**3,539,103,578**	**202,003,601**
180	min	3,079,590,117	201,111,331
	max	3,502,093,167	933,050,447
	avg	**3,070,942,248**	**511,007,813**

Table 3. The expensive input minimum, maximum and average cost of the 20 repetition for each budget. The cost of processing `graphviz` inputs grows super-linearly as the bound on input size grows. `flex` times out even on the smallest input size bound, so larger inputs do not lead to longer execution times.

Similarly, there are cases where it takes TreeLine the whole hour in finding new maximum input in the case of `flex`. This is misleading, however, because TreeLine quickly finds an input so costly that it triggers a timeout in the monitoring. The maximum cost that does *not* trigger a timeout likely varies due to uncontrolled variables like the host machine load state.

The minimum, maximum, and average maximum-cost found within each budget is shown in Table 3. The costs obtained for `flex` are consistent with conclusion on the time variation it takes TreeLine to find a maximum input (Figure 21b). For all budgets TreeLine reaches the same execution cost due to timeout. For `graphviz` the costs obtained show that there is a super-linear increase in cost when doubling the budget from 60 to 120. However, as we reach the higher budgets (i.e, 180) TreeLine's start to deteriorate. The pattern described next may explain the deterioration.

Consistently finding the same pattern. In Section 4.2.1, we illustrated that the maximum cost input found by TreeLine for `graphviz` with a budget of 60 is one that forms a cyclic digraph embedding a complete bipartite digraph on a subset

of its nodes. Following the same input selection criteria, TreeLine is capable of finding an input with the same pattern given a budget of 120.

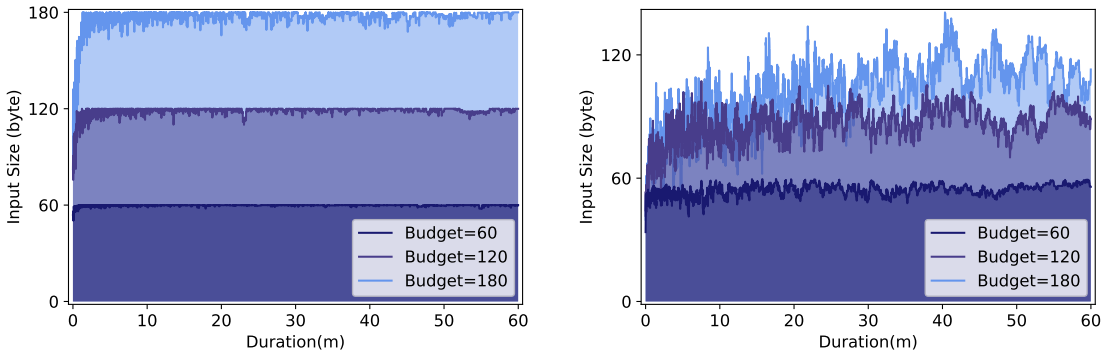
```
digraph{s:n,O:se,D,i,6,A,1,tH:n,M:w,_,T,J,1,f,P,A,q,R,Q,n,Q,Y,u,↩
    ↪9,0->7:e,K,u,j,J,V,S,1,6,q,P,O:se,N,Y,c,a,D,Q,K,0,N,W,J}
```

In the case of the 180 budget, the maximum cost pattern varies slightly from the pattern found with smaller length budgets. TreeLine find closely similar complex pattern by using sub-graphs (shown below), but it wastes some opportunities by over using the keyword `subgraph` which wastefully consumes its length budget.

```
digraph{f:ne->subgraph W{}subgraph I{}->subgraph I{{h=Qsubgraph ↩
    ↪r{}->J:c,6:sw,m,g,r,Y,U,d,2:s,E:ne,n:s,t->o:n,t,F,Fo:n,3↩
    ↪,C,C,V,d:s,9:w,36:sw,V,0:n2:s,C,2->N:c,6:sw,GQ,4,C,m,Gg}↩
    ↪G}d}
```

Exploring longer inputs. The length of the input generated, as discussed before, has a significant effect on the cost. Expensive patterns can be represented in shorter inputs. However, the budget set by the user implies an expectation that the given input budget is needed to represent interesting and unknown patterns. Therefore, we designed TreeLine's reward function to push the search toward paths that maximize the input length. We expect TreeLine to make use of the whole length, or nearly so (but without wasting it).

In Figure 22 we show the median size of generated inputs over the allowed time. For each different budget a search within the maximum possible input length maximizes the chances of finding an expensive input within the desired budget. In the case of `graphviz` (Figure 22a), TreeLine hits the maximum possible input-length within the first few seconds of search. However, in the case of `flex` (Figure 22b), TreeLine consumes only part of the budget of 120 or 180 bytes.



(a) `graphviz` (b) `flex`

Figure 22. The median length of generated inputs across 20 repetition for each budget over the allowed time. TreeLine quickly exploits its full length budget.

This again appears to be an effect of quickly finding an input so expensive that it times out.

Overall TreeLine appears robust with at least a moderate increase in length allowance, from 60 to 120 bytes, on these examples. At triple, it remains effective but shows signs of needing refinement.

4.3 Discussion

4.3.1 Threats to Validity. Our evaluation compared to PerfFuzz (RQ1) is a question about approaches and algorithms, not implementations. However, as usual our evaluation can only compare representative implementations of the approaches. Implementing TreeLine atop AFL goes some way toward facilitating a meaningful comparison to PerfFuzz as the leading representative of the mutational approach, but it is far from perfect. We know that TreeLine executes far fewer test cases per unit time than PerfFuzz, but we cannot say with confidence how much of that difference is attributable to the basic approach (TreeLine expends more computational effort choosing which search node to explore further) and how much is attributable to the TreeLine search running in

a separate Python process and communicating with the AFL-based test harness through inter-process communication, while the PerfFuzz implementation is integrated into the AFL process. A variety of other incidental implementation decisions might affect the results.

More fundamentally, both TreeLine and PerfFuzz depend on provided artifacts: a grammar in the case of TreeLine, and seed inputs in the case of PerfFuzz. We have tried to use "minimum effort" artifacts in the evaluation — a single example input as a seed for PerfFuzz, and a grammar derived directly from documentation or converted from a parser specification file for TreeLine. In the case of `flex`, we provided PerfFuzz a set of seed inputs that included all the constructs in the grammar used by TreeLine. While we have tried to be fair (and in particular, not to "tune" grammars to improve TreeLine's relative advantage), results could certainly be different with a different set of grammars or seeds.

Comparisons of tools that require even a small amount of human configuration can be done only with a limited set of examples, which may or may not be representative of a larger population of applications. We have selected two examples (`wf` and `libxml2`) that were previously used in evaluation of the PerfFuzz approach, which we believe are representative of systems for which the mutational approach should be advantageous. Moreover, we introduced two new examples (`graphviz` and `flex`) which we believe are representative of applications driven by syntactically richer input, and therefore more favorable to the TreeLine approach. A larger and more varied set of examples would certainly be desirable, but will take time to accumulate.

4.3.2 Alternatives. We cannot say how much of the difference between the performance of TreeLine and that of the best current mutational

fuzzers is specifically due to the MCTS approach, and how much is due to using a grammar. It would be useful to pick apart those two changes, but adapting any grammar-based fuzzer to search for performance bugs will require concomitant changes to other parts of the search strategy.

Two weaknesses of TreeLine might be addressed by techniques that have been developed in grammar-based fuzzing. One is that, while a context-free grammar greatly reduces the proportion of syntactically invalid inputs generated, in many cases they remain the majority of generated texts. This has been addressed by grammar-based fuzzing with additional constraints, notably in fuzzers for programming language texts (Mathis et al., 2019; Srivastava & Payer, 2021; Vyukov, 2021). Additionally, some grammar-based fuzzers perform mutation operations on a concrete or abstract syntax tree (e.g., replacing a subtree). This would not be compatible with the MCTS search strategy, which requires *all* of a nodes children to be populated together, but an adaptation might be possible.

Throughout the evaluation process, we measured the length of generated inputs in bytes. This cost choice allowed us to make a fair comparison to PerfFuzz, but is not the choice we would have made otherwise. For applications driven by richly structured input, efficiency is unlikely to be determined primarily by reading and scanning the input; a higher level measure such as number of tokens is likely to be better.

Producing grammars for use with our prototype tool is considerably easier than writing a grammar for a parser. It does not have to be LALR(1) or LL(k) or even unambiguous. We found it short work to transcribe a Yacc parser grammar in `graphviz` to an acceptable form, and to compose a simple grammar for `flex` from a page of documentation.

Nonetheless, it might be better yet if we could infer grammars directly from program behavior. In principle, grammars for TreeLine can be inferred from seed inputs. Unfortunately this does not work very well at the current state of black-box grammar inference engines. (We have not tried white-box or grey-box grammar inference engines such as Mimid (Gopinath, Mathis, & Zeller, 2020; Höschele & Zeller, 2016)). We attempted to use TreeLine with Glade (Bastani et al., 2017), a state-of-the-art tool for grammar synthesis. Although Glade succeeded in synthesizing grammars for all benchmarks, they were not suited for generating performance tests with TreeLine. For example, the grammar synthesized for `libxml2` using Glade tended to be limited to the smallest set of XML tags (e.g. `<a>` and `</a>`) to produce valid XML inputs. The grammars generated for `graphviz` and `flex` were much larger than those we created by hand, and both included unbounded derivation cycles that our tool is not currently able to handle.

4.3.3 Other Related Work. TreeLine as well as many alternatives fit generally into the field of *search-based software engineering* (Harman, Mansouri, & Zhang, 2012). The most closely related work is mutational fuzzing to find performance bugs and grammar-based fuzzing for other purposes, especially for security. SlowFuzz and PerfFuzz are representative of the state of mutational fuzzing for performance testing. As PerfFuzz dominates SlowFuzz through introduction of multi-objective search, and is otherwise similar, a separate comparison to SlowFuzz is not useful. Related work in grammar inference and grammar-based fuzzing has been noted above.

Syntactic structure is not the only kind of structure that one might use as a framework to search for pathological inputs. The subfield of combinatorial test case generation searches for combinations of *values* that trigger bugs. GA-Prof

uses a genetic algorithm search to find application bottlenecks (Shen et al., 2015); Forepost uses unsupervised learning to search primarily for a set of boolean choices (Grechanik et al., 2012). These search spaces are very different from a search of syntactic structure, and likely require different techniques.

Glass-box analysis, typically using symbolic execution, has also been applied to characterizing worst case execution time, which could be viewed as subsuming search for pathological inputs (B. Chen et al., 2016; Zaparanuks & Hauswirth, 2012). These are powerful techniques but usually limited to smaller components of an application due to their complexity.

CHAPTER V

PerfRL: FINDING ACCURATE PATHOLOGICAL TEXT CASE PATTERNS WITH REINFORCEMENT LEARNING

Machine learning has been influential in many different domains but has barely been explored for performance analysis. Our comprehensive search for techniques that explicitly adopt machine learning to generate input for performance testing resulted only in work presented by Grechanik et al. (Grechanik et al., 2012). As presented in the related work (Chapter II), Grechanik et al. (Grechanik et al., 2012) approach did not generalize enough to identify inputs automatically.

As machine learning proves itself a good or promising solution for different applications, we thought it important to explore what established machine learning modules apply to the input generation problem. Finding an arrangement of inputs that maximizes a method's execution is a very high-dimensional problem. Machine learning is widely known to be suitable for high-dimensional problems.

Machine learning techniques are classified into supervised, unsupervised, and reinforced learning. We first evaluate the applicability of each learning method then discuss the most appropriate method for the domain of input generation.

Supervised machine learning approaches are applicable in cases where labeled data are available. In software analysis, data is not an issue as inputs can be paired with the runtime cost based on a given traced run. In fact, the sheer amount of data where slight changes in input could produce a slight uninteresting change in output is one of the problems. Another problem is that supervised learning models must use random data generation methods to generate actual new inputs. Otherwise, the model would be passive in that it looks only at what user provided as inputs rather than finding new ones.

We think that supervised learning methods are more applicable in performance-driven source code learning than input generation. For example, using regression-testing techniques that identify the source code change causing a performance change (Luo, Poshyvanyk, & Grechanik, 2016; Sandoval Alcocer, Bergel, & Valente, 2016) can create labeled data of source code fixes for performance improvements. A challenge here is to automate the deployment, run the application, and collect traces of heterogeneous artifacts and many different versions of the same artifact.

Unsupervised machine learning techniques, where no labels are available, are suitable for clustering, anomaly detection, or finding associations amongst other applications. Similar to supervised learning, unsupervised learning can cluster expensive inputs from cheap inputs. However, also similar to supervised learning, there must be an input generation used to explore new paths. Depending on user-created tests would lead to limitations such as those presented in different passive performance analysis approaches (Ball & Larus, 1996; Graham et al., 1982; Nistor, Song, et al., 2013).

Reinforcement learning maintains a reward function that increases as a learning model gets closer to the optimal solution by trying different inputs at different learning phases. For input generation, the reward function can be seen as a fitness function that increases as a method's execution cost increases and decreases otherwise. At each learning phase, the reinforcement learning model should learn to generate expensive inputs. Therefore, we investigate the applicability of using reinforcement learning to generate pathological inputs of diverse applications.

We covered the essentials of reinforcement learning in the Background Chapter (Section 3.3). Similar to adopting Monte Carlo Tree Search, we use context-free grammar (CFG) to generate valid inputs. The fundamentals of CFG are covered in Section 3.1.

5.1 RL Integration Formalization

Similar to the Monte Carlo tree search (MCTS) based, we use Context-Free Grammar (CFG) to search for pathological inputs using reinforcement learning (RL). We adapt the same notion of cost ans budget for grammar and derivation as defined in Section 4.1.3. However, options , or *actions* as they are referred to in the reinforcement learning domain, require additional special handling to work with reinforcement learning.

To define the *actions* an agent can take we need to consider the number of possible options from each production rule. Different CFG have different number of production rules and a different number of alternations for each production rule. In our work, we abstract every aspect of the environment (e.g., states or rewards) to be exchangeable and allow for different experimentation methods. However, for simplicity, consider the `left-hand side` of any production rule as the *state* the agent is in at any given moment. And the alternations (options) on the `right-hand side` of any production rule are the possible *actions*.

The issue that arises from the simple definition of *states* and *actions* is that any reinforcement learning model must have a well-defined *state* and *action* space. As the number of alternations for each production-rule might vary, this definition needs some constraints. Thus, we introduce the notion of *valid-options* for each state. Assume we have a grammar of two symbols $\langle A \rangle$ and $\langle B \rangle$. Also, assume $\langle A \rangle$ has two alternations, and $\langle B \rangle$ has three. In this case, we say the number of possible

actions from any state is 3. However, the number of *valid-options* for $\langle A \rangle$ and $\langle B \rangle$ are 2 and 3, respectively. This definition means the number of possible Q-values at any state in our neural network equals the maximum number of alternations from all production rules.

It is important to note that it is valid in reinforcement learning to allow an agent to take non-existing options for cases such as ours, where the number of possible actions varies based on states. In such design, the environment ensures nothing changes in terms of the *state* if a non-existing action was taken, then we can re-introduce the agent with the same options again. Such design means there will be many wasted computation efforts before the agent can change its choice either through exploration or exploitation. We prefer to have more control over the *actions* the agent can take to avoid such wasted computations. Thus, we limit options from any state to the set of *valid-options* only.

The notion of *cost and budget* we formalized earlier along with the notion of *valid-options* together restrict the possible *action* from any state. Instead of only constraining the *valid-options* dynamics on each production rule's available alternations, we also tie it to the *remaining-budget* as we did with TreeLine. After each step, we reevaluate the *remaining-budget* within the environment and constrain the *valid-options* according to the *remaining-budget*.

Our definition of the agent's *actions* demonstrates that we treat the grammar elements with no bias and safely interact with the agent without infinite derivations. Therefore, if the grammar thoroughly expresses the target application input, and the agent is capable of learning, then the elements of importance will be learned in the training process itself.

Another parameter we need to formalize is the rewards function as it plays an instrumental role in training an agent using reinforcement learning. For each step the agent takes (moving from one state to a new one), we have to send the agent a feedback to help it assess the quality of the move it made. As described in Sections 4.1.2 & 4.1.5.2 from TreeLine, the feedback from running the application with a new input is diverse based on target application and the range is of minimum to maximum is unknown. Similar to TreeLine, PerfRL use the the feedback from AFL (Zalewski, 2013) to get the control flow graph hit rates for each input (i.e., NewCost). However, we do not utilize the NewCov or NewMax in the case of PerfRL.

Moreover, we abstract the cost obtained using a defined reward method as we abstract many other parts of our environment (e.g., state representation). The abstraction of reward methods allows for different representations of the reward; hence, different experimentation opportunities. An example reward method abstraction is a method that performs a few warm-up runs of the target application to establish a regular input average cost (total edge hits). Once we find the average, a rewarding method can be the percentage increase or decrease in total edge hits relative to the average input cost. We elaborate on different examples of reward functions in Section 5.2.2.

Using CFG as a driver for input generation implies we lose the previously generated derivation context at each new state. For example, at some intermediate step in the derivation process, we have a sense of the non-terminal symbols to come as they are preserved in the stack. However, we do not know the sequence of decisions made to reach the current state. For the agent to learn how to generate the most expensive input, it must keep track of past experiences and choices it

made. Given that our environment is essentially an interpreter of the grammar, the best option is to use the call-stack as a state representative to help the agent perceive the derivation choices it made and what lies ahead. However, again, the neural network for reinforcement learning must be defined ahead of training. By definition, the call-stack is dynamic. Thus, we cannot predict the maximum number of elements the call-stack might contain during the derivation process.

From the essentials of reinforcement learning (Section 3.3), we stated that a problem must be formalized as Markov Decision Processes (MDP) to solve it using reinforcement learning. One of the MDP properties is to have a full observation of the environment's state. However, from the paragraph above, we see that we cannot observe all the states' information in our problem domain. We lack the observation of essential information about the context of the derivation. Therefore, we define our problem as a Partially-Observable Markov Decision Processes (POMDP). For problems defined as POMDP, a possible solution is to infer the missing information using the neural network itself.

In most of the published work in reinforcement learning, problems are formulated as MDP problems. Therefore, reinforcement learning literature uses linear or convolutional neural networks to demonstrate the training process and model. However, in the domain of Natural Language Processing (NLP), where a context is needed to classify an input for tasks such as machine translation or text generation, linear or convolutional layers are not suitable. NLP problems are posed as a supervised learning problem where labeled data is available. To carry the context, say between generating one word after the other in machine sentence generation, NLP techniques uses Recurrent Neural Networks (RNN) for training.

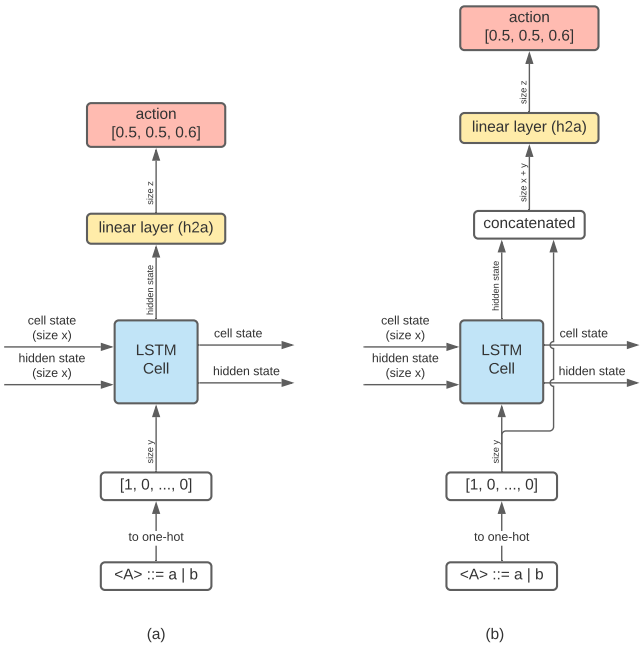


Figure 23. Two neural network prototypes using LSTM to represent context of derivation. Each prototype is a step snippet of the neural network structure in the possible chain of derivations.

RNN are a special type of neural networks that carry a hidden state between the steps of generating an input. The hidden state main purpose is to maintain context between steps of the same episode. For example, the hidden state of RNN could be carrying the subject-verb agreement information observed in the first step to be used in some later step while generating a sentence. RNNs are the basic neural cells of recurrent neural networks. They tend to carry information of recently seen contexts. As the context gets larger (i.e., longer sentences), RNN suffers from recalling long seen information. Long-Short Term Memory (LSTM) cells are introduced to mitigate the issue of larger contexts. LSTM is an enhancement on the original RNN cells to carry recently observed information (hidden-state) as well as ones seen much earlier (cell-state) in any context.

Although not much work has been done in reinforcement learning using LSTM cells, we believe it is a good fit for our problem as it has been successfully used on other closely related domain such as NLP. To mitigate the partial state observation problem, we will rely on LSTM cells to carry the information and influence the agent choices based on the derivation context. Figure 23 shows prototypes of how we think LSTM cells can be used in our domain. See that both versions (a) and (b) uses linear layers to map states to the number of possible actions as we described earlier in this section. We believe we should use as many linear layers as needed to fit the problem. Hence, using the LSTM cells does not mean we do not use other forms of neural nets.

5.2 Solution Design and Abstractions

In this section, we will describe the broad implementation design decisions (Section 5.2.1), describe the state and reward abstractions we use (Section 5.2.2), and discuss our neural network structure (Section 5.2.3).

5.2.1 Design Overview. The general implementation and design decision of PerfRL follow the conventional Deep Q-Network (DQN) design (see Section 3.3 in the background chapter for details) . For example, as shown in Figure 24, we have a neural network model, an agent, a replay memory, and an exploration policy the agent follows. However, some of the details of each component are different from what have been proposed in essential DQN literature (Lillicrap et al., 2015; Mnih et al., 2016; 2013; 2015; van Hasselt et al., 2015). In general, we are either modifying the process to be compatible with our problem or apply optimizations specific to our problem. In the rest of this section, we explain the complete process of generating an executable input (an *episode*) and detail how the components interact within an *episode*.

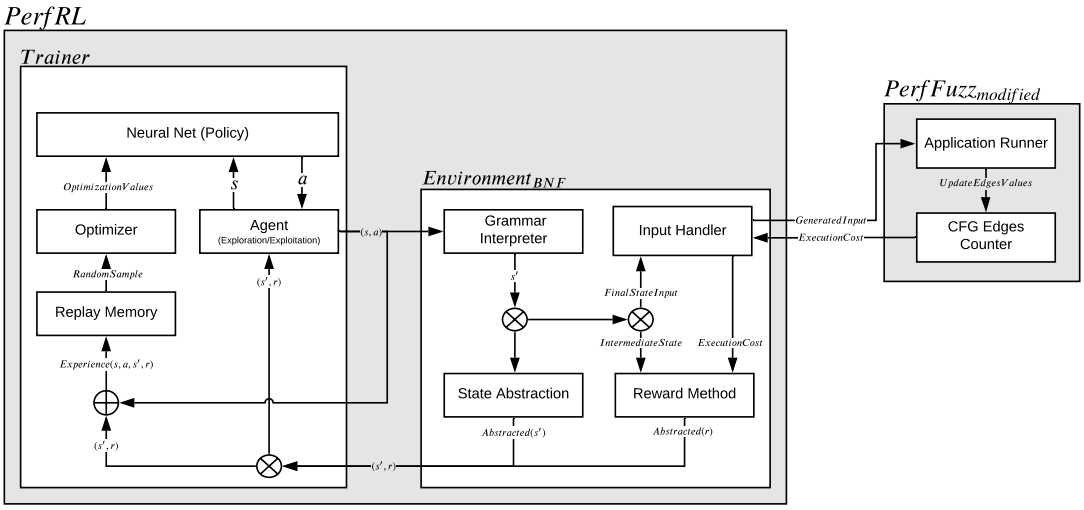


Figure 24. PerfRL high-level architecture and interactions.

A training session in PerfRL starts with a warm-up period. We do not depict the details of the warm-up phase in Figure 24. In the warm-up phase, we use the environment to generate n number of executable inputs randomly. We use the collected costs to generate future rewards. There are different ways in which we use the collected costs depending on the training session (Section 5.2.2). Similar to TreeLine, the main advantage of a establishing a base of cost is to scale possible high absolute cost to some appropriate known scale. For example, passing an array $[9, 8, 3, 4]$ to QuickSort could result in cost $c = 3391$ where we might be targeting cost in range between 0 and 1.

Absolute execution-cost values are not suitable for rewarding the agent. The absolute value of a trivial input such as the one we showed here will have very close proximity to the worst-case input execution-cost value. Moreover, they are both positive. Thus, they might not categorically express the difference between a worst-case input and any other input. Furthermore, in reinforcement learning, rewards tend to be small (between -1 and 2) to avoid causing a sudden increase or decrease

in *q-values*. Thus, we would like to use a relative value of the absolute execution-cost. The warm-up phase's total cost could give us a notion of the average costs. Hence, allowing us to find different rewards method based on the established average rather than the absolute execution-cost.

With an established sense of average cost within the environment, the agent starts to interact with the environment for learning. We define a *step* as each choice the agent makes within the grammar for which it takes it from one derivation step to the next one. An *episode* is the collection of *steps* starting from the root element in the grammar until obtaining a final input that we can pass to the target application. With these definitions, we explain the details of the process of each *episode*.

First, the agent selects an action a either randomly (exploration) or based on the current policy (exploitation) using a predefined epsilon-greedy strategy. We configure our epsilon-greedy strategy to allow very few exploration opportunities toward the end of completing any learning session.

Second, the current state s and the action a are passed to the *Grammar Interpreter* within the environment to take a *step*. As long as there are *non-terminal* elements within the derivation sequence, we do not have a final step. The *Grammar Interpreter* manages the status of the derivation. It classifies each step to be either an intermediate-step or a final-step.

Third, based on the step's type, the *BNF* environment will determine the reward type. It will keep passing passive reward back to the agent as long as we did not reach a final-step. Passive rewards can be defined as we see fit. We initially used the value of 0 for intermediate-steps, as it is not possible to measure the intermediate-step's quality without knowing the final-step it leads to. However,

we later used a reward of 1 for intermediate-steps as they encourage the agent to produce longer derivations than shorter ones. We think that longer derivation (selecting *non-terminal* options as often as possible) is the desired behavior in general for finding pathological input. It usually means the agent will consume the whole allowed budget. Also, we believe that longer derivations increase the probability of finding interesting input patterns than shorter ones.

Fourth, regardless of whether the step is a final or intermediate, the *BNF* environment will pass the reward and the current derivation state through a state and reward abstraction objects. There are many ways in which we can represent the state and the reward. Therefore, we thought of decoupling the interpreter's state and the execution cost from the state and reward observed by the agent and neural net. Different state representations and reward methods have different advantages; this setup allows us to experiment with more than one state representation and reward calculation for different sessions. We will elaborate on some of the state abstractions we developed and reward methods in Section 5.2.2.

Fifth, the new state s' and reward r are passed to the *Trainer*. One copy is fed back to the agent to take a new action, and another copy is assembled with the previous state s and action a to form an experience. Experiences are added to replay memory for future sampling and optimizations. At each *step*, the *Trainer* collects k random sample experiences to be replayed and use their values for optimization as typically done in reinforcement learning.

In PerfRL, we made a fundamental change to the experience notion from the known notion in reinforcement learning literature by collecting *episodic* experiences instead of *step* experiences. Conventionally, each *step* from a state s taking action a at time t and observing the new state s' and reward r is an

experience (i.e., $Experience = (s, a, s', r)$). In the case where the problem is formulated as an MDP, this notion of experience is valid. However, in our case where the problem is formulated as POMDP, we either have to add the hidden-state (hidden-states are defined in Section 5.2.3) to each experience or look at a complete *episode* as an experience.

It is possible to capture the hidden-state within the *step* experiences. However, we cannot guarantee an old hidden-state's integrity compared to the policy's future value. Therefore, we define the experience as the ordered collection of *step*s within an *episode*. If at *episode* e_1 we observer the *step*s $p_1(s, a, s', r), p_2(s, a, s', r), ..., p_t(s, a, s', r)$ where p_1 is the first *step* in a derivation from the root element and p_t is the last *step* where a final input is generated, then the experience E for *episode* e_1 is $E[p_1(s, a, s', r), p_2(s, a, s', r), ..., p_t(s, a, s', r)]$.

By sampling episodic experiences and optimizing the policy at the end of each *step*, we complete a full *step* cycle. Our *Trainer* reset the environment to its initial state by the end of each *episode*. Hence, even for optimization, where we replay stored experiences, we reset the environment. Each training session has a predefined number of *episodes* to run. The final policy is the neural net state by the end of running the last *episode*.

5.2.2 State and Reward Abstractions. One of the implementation and design challenges we had to address is representing the states and rewards to be perceived by the neural network. Any neural network expects a numerical representation of the environment state. There are many possible ways and indicators for state representation. With no clear advantage of one state representation over the other, it is only realistic to separate the *Grammar Interpreter*'s state from the state sent to the neural network.

The same idea applies to the execution cost value. We briefly stated that the absolute execution-cost value is not suitable for reinforcement learning. As explained in the warm-up phase, we can initialize a normal cost average that we can use for later reward calculation. Additionally, we also can track an upper and lower bound of observed execution costs for later reward calculation. Decoupling the actual execution cost from the reward function allows us to try different rewarding methods.

We only explain the relevant state and reward-method abstractions that we use in the evaluation (Section 5.3). However, we explain the most simple ones to introduce the abstractions elements. We discuss the state-abstraction design first.

We can use many indicators from the *Grammar Interpreter* to represent an abstract state. For example, the derivation sequence, the remaining budget allowed for the input, and the production rules are all indicators. In general, we think the sequence of derivation steps or the current production rule are the most useful ones. The budget is valuable, but the remaining budget's value might be incomprehensible to a neural network as it is to us, humans. Also, even for the derivation sequence steps, we cannot represent the full sequence. As we explained before, the number of derivations cannot be bound even with a defined budget. At the same time, the neural network expects a well-defined set of elements within a state. Thus, we can only represent a defined buffered size of the derivation sequence.

Despite these challenges, we have a state-abstraction that captures each idea. However, the one we use here is the simple production rule state-abstraction. We use a one-hot representation to express the production rule we are evaluating at

any moment. Moreover, we include all the available options for all production rules. For example, for a grammar as the one given below.

$\langle A \rangle$ ::= $\langle A \rangle$ $\langle B \rangle$ | 'a'

$\langle B \rangle$ ::= 'b'

The production rules state-abstraction when we are evaluating the production rule $\langle A \rangle$ will be $[1, 0, 1, 1, 0]$. Similarly, when we are evaluating the production rule $\langle B \rangle$ the state-abstraction will be $[0, 1, 0, 0, 1]$. Any time we evaluate a production rule, we highlight its position within the array and the positions of all its options with 1 while all other positions are set to zero.

This production-rules state-abstraction highlights the production rule we are evaluating at any given moment to the neural network. In other words, it indicates where the agent is standing. What we are missing here is the context of previous derivations. However, as stated before, capturing the full context is computationally infeasible. We can either settle for partial context representation using the buffered sequence of derivation state-abstraction or delegate the context's inference to the neural network. With the production rule state abstraction case, we choose to delegate the context to the neural network.

For the reward-method abstraction, there are two main ideas. The first option is to keep a bound of upper and lower execution-costs. The upper-bound is defined as the maximum absolute execution-cost the agent observed up until any given moment. Furthermore, the lower-bound is defined as the upper-bound minus k, where k is a defined threshold based on the target application itself. The second options is to initialize a normal execution-cost average based on random input mutations.

We can use the first approach to find the normalized value of the current input execution-cost based on the defined upper and lower bounds. The issue with this approach is its introduction of the moving target notion. As the agent observes new inputs, what is know to be an expensive input in the past might not be expensive in the future. Intuitively, this approach might seem advantageous as we want to drive the agent toward more expensive inputs to gain positive knowledge about the program space. However, based on our experience, this approach introduces significant instability in the learning process.

Therefore, we tend to experiment with the second approach, where we define a fixed average of normal execution-cost then use the average value to calculate future rewards. One of the reward-methods defined to use the found average calculate the new input's execution-cost percentage increase or decrease relative to the defined average. Thus, any input that costs more than the defined average will result in a positive reward been sent to the agent and vice-versa.

5.2.3 Neural Networks Structure. A unique characteristic of our problem compared to ones described in reinforcement learning literature is the inability to observe the environment's full state. Thus, we defined our problem as POMDP. To overcome this challenge, we looked into the popular neural network used for natural language processing in supervised learning settings. Moreover, we conjectured that LSTM cells could help in inferring the missing context of derivation sequences. Here we show how we use LSTM cells within our neural network model.

The neural network structure illustrated in Figure 25 shows the basic building blocks of our neural network. The depth and width of linear neural networks, as well as the size of the hidden-states, might vary based on the target

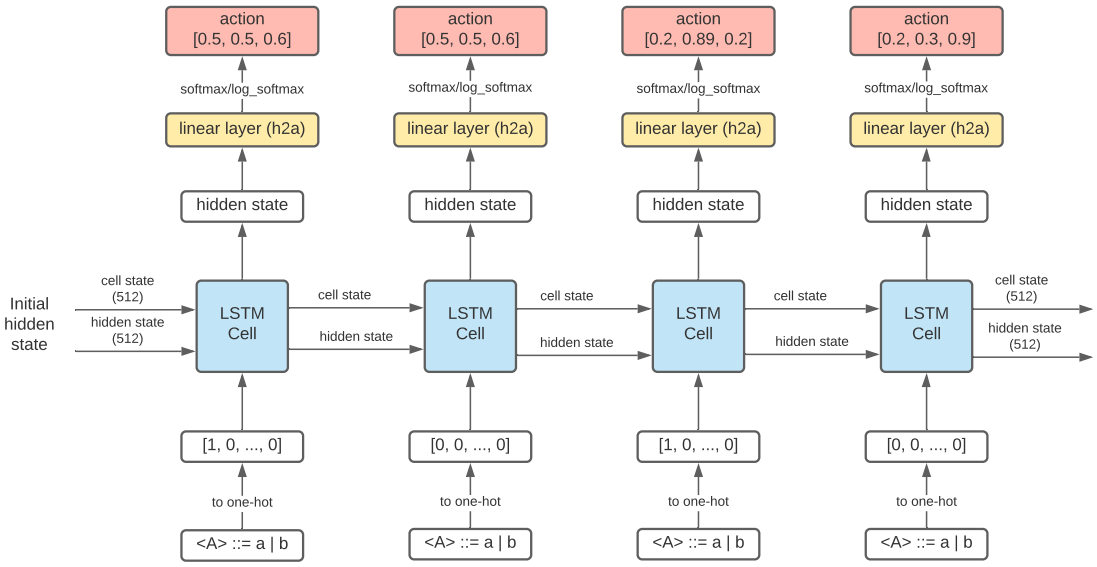


Figure 25. PERFRL neural network basic structure. It uses LSTM cells along with linear neural network to accept the input from a state-abstraction and outputs and action based on the *valid-options*

application and grammar. However, the illustration highlights how the LSTM cells carry the context from one step to the next for each *episode*. Each time we start a new *episode* of derivations, we start with the same valued hidden and cell states (usually zeroed). The LSTM cells' goal is to learn how to represent the context of derivation as we step within an *episode*. Hence, it passes the learned values of the hidden and cell state from one LSTM cell to the next as the agent makes the derivations. At the same time, we use the same hidden-state from each LSTM cell at each step to influence the decision of which action to take from the current production rule.

5.3 Benchmarks & Experiments

The ultimate goal is to demonstrate our approach using multiple and diverse target applications. However, to better understand how effective our approach, having a small set of target applications with known possible behavior

is more valuable at this stage than broadening our test to several and larger target applications. At the same time, we acknowledge the risk of fine-tuning the approach to fit the smaller set of a target applications. To this extent we focus our testing on QuickSort (GeeksForGeeks, 2021)[1] and WordFrequency (de Barros, 2021).

As know to many, the QuickSort worst-cases inputs are either a sorted list, reversely sorted list, or a list of equal values. The WordFrequency, on the other hand, is a simple word counter application. It takes text as input and returns a count of each word in the text. One of the known worst-case inputs of WordFrequency is an input where a word is repeated as many times as possible. The repeated word input causes it to hash the word to the same bucket. Hence, introducing a hash collision. `PerfFuzz` reports the following input as the one it finds for the worst-case execution.

```
t <81>v ^?@t <80>!^?@t <80>!t t^Rn t t t t t t t t t
```

We use the grammars shown in Grammar 3 and Grammar 4 for QuickSort and WordFrequency respectively. We recognize that the grammar written for WordFrequency is a much-simplified version of what a program author would write or a synthesizer would generate. We effectively test on other comprehensive grammar, but we use the grammar given to show sample results.

We run every experiment with a budget of 10 given a character-based defined cost. Because the terminal elements are composed of a single character in both grammars, the maximum possible string length is equal to the budget. Due to space limitations, we cannot show all possible plots that would help illustrate our

[1]We extended the QuickSort implementation to read inputs from file instead of `stdin`.

⟨*entry*⟩ ::= ⟨*entry*⟩ ⟨*digit*⟩ | ⟨*entry*⟩ ⟨*space*⟩ | /* empty */

⟨*digit*⟩ ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

⟨*digit*⟩ ::= ' '

Grammar 3. Manually written context-free grammar for QuickSort algorithm. Used in evaluating PerfRL.

⟨*Word*⟩ ::= ⟨*Word*⟩ ⟨*Char*⟩ | /* empty */

⟨*Char*⟩ ::= ⟨*T*⟩ | ⟨*Space*⟩

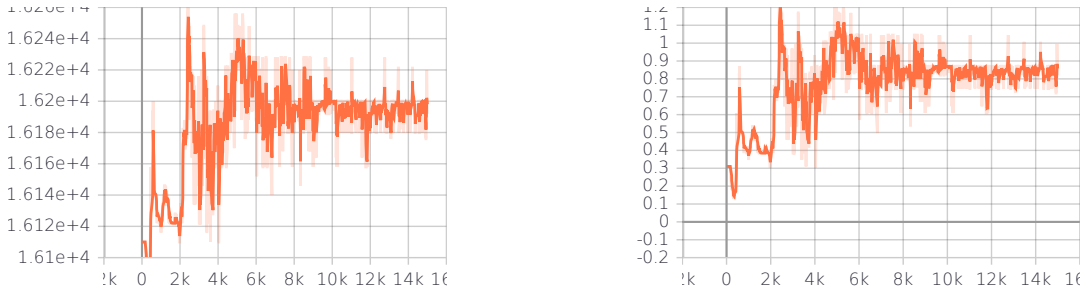
⟨*T*⟩ ::= 't'

⟨*Space*⟩ ::= ' '

Grammar 4. Manually written context-free grammar for WordFrequency application. Used in evaluating PerfRL.

model's current state. Therefore, we will briefly present the final results for each target application and then focus on one of them to highlight evaluation details.

PerfRL can confidently learn to generate one of the expensive worst-case input of QuickSort. In particular, our model learns to generate an array of the same digit (e.g., `9 9 9 9 9`). However, for the WordFrequency application, our model usually converge on a sub-optimal input (e.g., `tt t`). The WordFrequency example is a more interesting and challenging one; thus, we elaborate on its result. Figure 26 shows the learning trend of the agent on the WordFrequency. The values shown are smoothed values for readability. We can infer from the charts that the agent finds the most expensive input around the $2,000^{th}$ episode (known actual cost is $16,284$) but does not sustain the knowledge. At the same time, it does not converge on a significantly bad policy. The range in which the policy converges is between 0.8 and 1.0, which is between $16,188$ and $16,228$ of the actual cost.



(a) Actual Cost (b) Rewards

Figure 26. Policy-based results of training PERFRL for 15,000 episodes on the WORDFREQUENCY. The actual cost is the total edge count of executing an input. And the reward is what the agent receives as a reward for the actual cost.

The plots alone might not help understand what the agent was observing. Thus, we collected all the generated inputs then added to replay memory. These inputs influence the learning as they are what the agent observe and use to adjust the policy. However, toward the end of the experiment run, and because we decrease the exploration opportunities given an epsilon-greedy strategy, these inputs can reflect what the policy learned. Therefore, they are a combination of the learning opportunity and an observation of the learning progress. Table 4 shows the top-10 results sorted by count (how many times they were observed) and Table 5 shows the top-10 results sorted by cost (the actual cost of Control-Flow Graph edge count). We discuss the limitation of our model in the next section, given the results shown here.

The most important takeaway to note here from the result shown in Tables 4 and 5 is the ability of our model to explore. Good exploration is a significantly important factor in learning. Without the ability to form interesting patterns of input, the agent will have no means to learn. The results shown in the table indicate that our exploration strategy yields different variations of the most

expensive input. The goal is to increase the generation of these expensive inputs compared to less costly ones.

Input	Count	Cost	Reward
‘’	1,550	16,042	-0.066
‘tt tt’	1,303	16,180	0.793
‘t’	1,019	16,092	0.245
‘tttt tt’	844	16,182	0.806
‘tt t’	707	16,200	0.918
‘tttt t’	705	16,188	0.843
‘tttt t ’	684	16,188	0.843
‘ttt tt’	653	16,188	0.843
‘ttttt ’	646	16,140	0.544
‘tt’	553	16,094	0.257

Table 4. WordFrequency top 10 inputs arranged by count (how many times they were observed) using PerfRL.

Input	Count	Cost	Reward
‘t tt t t t’	2	16,284	1.441
‘tt t t t t’	3	16,284	1.441
‘t t t t t’	1	16,276	1.391
‘ttt tt t t’	3	16,258	1.279
‘tt ttt t t’	3	16,258	1.279
‘t ttt tt t’	3	16,258	1.279
‘t ttt t tt’	1	16,258	1.279
‘t tt ttt t’	1	16,258	1.279
‘ttt t tt t’	3	16,258	1.279
‘ttt t t tt’	2	16,258	1.279

Table 5. WordFrequency top 10 inputs arranged by cost (the actual cost of Control-Flow Graph edge count) using PerfRL.

5.4 Limitations & Challenges Discussion

It is hard to critically analyze PerfRL's performance at this stage, given some identified limitations. Instead, we address these limitations for future research. We have identified four main limitations of our technique. First, wasted effort in exploring the input's length instead of its structure. Second, frequent

and fast policy adjustment. Third, possible worst-case input structure variations. Finally, memorizing a worst-case input instead of learning its pattern. We will discuss each challenge in detail next.

Exploring Input Length. The last statement we made about our model was about its ability to explore the input space and finding interesting input patterns. However, the exploration as it works is not ideal. From the results in Table 4, we can see that the exploration strategy produces an empty string $1,550$ times as the most frequently seen input. The learning rate charts in Figure 26 shows that the policy determined that longer inputs are more desirable than shorter ones very early in the learning process.

Nevertheless, the agent generates shorter inputs more frequently based on the exploration strategy. Given the grammar and budget, the agent would have to make ten decisions about the length of the input before it has an opportunity to decide on the structure of the input. With a high epsilon-greedy value, it is most likely that the exploration will fall within the length steps rather than the structure deciding steps. The frequent exploration of length wastes many opportunities of exploring the interesting problem of structure.

To mitigate this issue, we introduced an amended exploration strategy. Instead of relying only on the value of the epsilon-greedy strategy to decide to explore or exploit, we also randomly evaluate the difference between the two highest Q-values according to the policy. Therefore, if the epsilon-greedy value is very high, and the policy is not $k\%$ confident about the next step, we will most likely explore. Otherwise, we exploit the policy. Based on our experiments, we found that it is hard to find a reasonable k threshold for good exploration opportunities across different derivation steps.

Another possible future mitigation is to force longer derivation from the grammar interpreter itself. Given a budget and some grammar, we evaluate the remaining budget and the production rule in hand for each derivation step. If we have enough remaining budget and the production rule options could produce zero or more elements, then we favor options that produce more elements over the null production option. Hence, the *valid-options* should be amended to favor longer derivations. Therefore, whether the agent is exploring or exploiting, it will only be allowed to elongate the derivation whenever possible.

Fast Policy Adjustment. Steady and gradual learning progress is one of the most important properties of any machine learning model. A model that steadily improves the learning rate is one that accommodates new positive experiences without forgetting older ones. As shown in Figure 26, our model indicates that it is sensitive to new experiences. We can observe the issue by looking at how much the learned policy fluctuates between one episode and the next.

We can attribute the fluctuation problem to high learning rates. We experimented with lowering the learning rate, which improves the stability of the learning progress. However, even with a low learning rate (compared to the practical values typically used), we still suffer from fluctuation.

Another factor could be the size (depth and width) of the neural network. A more extensive neural network is more capable of accommodating a complex problem space. Hence, it ought to stabilize the learning progress. However, a larger neural network requires extensive and diverse experiences. Increasing the number of experiences means we also increase the required learning time. Moreover, we have to guarantee the diversity of the generated samples. We can tolerate the required increase in training time. However, the diversity of the sample is coupled to the

exploration problem we discussed in the first limitation case. Thus, increasing the size of the neural network does not show effective results under the same exploration strategy.

We experimented with a neural network where we have two stacked LSTM cells instead of one. The result shows a significant improvement in stability. However, the learning progress becomes stable to a point where it is hard to change. Thus, within a few episodes, the neural network will converge on some policies that seldom change.

We see the fluctuation issue as a tuning problem. Thus, we use the best hyper-parameters setting we know and visit this as we address other challenges for more generalized tuning.

Input Variations. Different target applications might have different unique challenges. A common challenge we think all target application share is the possibility to have a variety of expensive inputs that lead to the same cost. For example, from Table 5 the inputs `ttt tt t t`, `tt ttt t t`, `t ttt tt t`, `t ttt t tt`, `t tt ttt t`, `ttt t tt t`, and `ttt t t tt` all share the same exact cost $(16,258)$. We think the possibility of having different inputs of the same cost is even larger than we could show here.

Different orders or shapes of input elements that lead to the same cost are expected behavior. However, these variations can make it even more challenging for the policy to converge on one input. A proposed solution is to unify these inputs that share the same execution cost under one form as they are added to the replay memory. However, it is not clear that this issue is the leading cause of converging on a sub-optimal policy.

Input Memorization. The only example we showed for experimenting with QuickSort clearly shows that the policy converges on one of the inputs the express the worst-case execution (`9 9 9 9 9`). However, upon closer analysis, we see that the policy memorized the input rather than learning the actual pattern. For example, if we test the policy by forcing the first digit choice (e.g. 2), the policy will not continue filling the remaining digits with the same value.

More in-depth analysis indicates that the policy memorize some input rather than learning a pattern. Our final goal is to learn to generate some final input. Thus, memorizing a worst-case input does not contradict our goal. However, learning a pattern and expressing it to the developer is more useful than memorizing an input. It will allow a developer to understand what pattern exactly triggers the worse-case execution.

CHAPTER VI

GENERAL DISCUSSION & FUTURE WORK

The discussions for each technique in Chapters IV and V are specific to each approach. This chapter discusses more general topics that are significant in the performance analysis domain and encompasses both techniques (TreeLine and PerfRL). We first discuss the challenge of measuring the effectiveness of any performance analysis technique in Section 6.1.1. Second, we discuss how the techniques we introduced fits within the performance analysis domain to assist software engineers in achieving their goals (Section 6.1.2).

Moreover, we provide an overall conclusion (Section 6.2) and identify possible future work directions based on the presented topics in this dissertation (Section 6.3).

6.1 Evaluation Validity

6.1.1 Efficiency Measurement. An apparent issue in performance analysis, in general, is the lack of a unified method for efficiency measurement (Sánchez, Delgado-Pérez, Medina-Bulo, & Segura, 2018). The ultimate goal for any performance analysis technique is to effectively help software engineers find actual and previously unknown performance issues. Mytkowicz et al. (Mytkowicz, Diwan, Hauswirth, & Sweeney, 2010) illustrate the severity of the efficiency measurement problem. In their work, they show how a set of Java profilers (*xprof*, *hprof*, *jprofile*, and *yourkit*) do not agree on a *hot* method under a unified benchmark and testing data. Such disagreements indicate that there must be at least three wrong profilers (Mytkowicz et al., 2010). In examining the core issues between the given profilers, they found them violating a fundamental sampling attribute. None of the

profilers collected samples randomly as they often collected samples at yield points. Therefore, each profiler would serve a different purpose.

When presenting their evaluation, authors of performance analysis tools follow different evaluation methods. Some authors present their findings and confirm them with the application's developers to measure efficiency (Graham et al., 1982; 2004). Others evaluate their efforts against tools with similar general goals but focus on other strengths such as overhead or coverage (Coppa et al., 2012). A more prominent efficiency measurement methodology (Curtsinger & Berger, 2015; Grechanik et al., 2012) compares results with other performance analysis techniques given a unified application under test but one that expresses the presented goal better. The last method can be unfair if mishandled because the goals of the two performance analysis techniques can be different. For example, Curtsinger and Berger (Curtsinger & Berger, 2015), who developed an application for finding performance improvement opportunities within a parallel program, compare its results to the ones produced by *gprof* (Graham et al., 1982). The goal for *gprof* was never to profile parallel programs. Thus, the comparison does not hold.

Grouped benchmarks for performance testing such as the DeCapo (Blackburn et al., 2006) benchmarks is also insufficient efficiency measurement. These benchmarks are not necessarily a realistic representation of real-world applications. In addition, these benchmarks could lead to more tailored solutions, as the goal becomes to find *new* performance issues from the same applications.

Performance analysis tools do not apply fixes automatically. Due to the tradeoff between preserving the program soundness and the potential of finding actual performance issues (Section 2.1.4), applying fixes automatically would

limit the changes to trivial fixes. Using the performance improvement percentage based on automatically applied fixes is a precise measurement for compiler-level optimization techniques. It shows the benefits of automatically applied fixes. However, adopting the same approach by performance analysis techniques would limit their effectiveness.

As fixes are manual, the performance improvement measurement cannot be precise for the performance analysis techniques. Humans' experiences and understanding can vary significantly. Thus, fixes applied for the reported performance issues could provide different speedup values depending on the developer's experience. Nevertheless, adapting and fixing the top k number of reported performance issues is a sound efficiency measurement method. Such parameter would bridge the gap in the performance efficiency measurement limitation. However, adapting such technique usually requires a much longer time as it depends on the application maintainers willingness to put the effort.

In our analysis, we try to minimize the impact of the fundamental differences between our work and comparable techniques. For example, given that the seed from which we start the search is a grammar alternative approaches start from raw seed inputs, we attempted to provide raw-input-based techniques with as comprehensive seed input as possible. Furthermore, we used the same cost base as with comparable techniques (control-flow graph edge hits) to have a unified cost definition.

An even more compelling solution for the input-generation approach is to use existing performance analysis techniques to measure their efficiency. The major input generation efforts discussed (Grechanik et al., 2012; Lemieux et al., 2018; Petsios et al., 2017; Shen et al., 2015) could delegate the analysis to an designated

profiler. For example, using tools such as *gprof* (Graham et al., 1982) to monitor the ranking change in performance issues between different methods given found input for each technique. Performance analysis tools might not serve a precise goal an input generator is targeting, but multiple profilers should cover and highlight different goals.

6.1.2 Assisting Software Engineers. In the introduction of this document, we briefly discussed the importance of the problem we are solving. Also, we presented other closely related techniques that try to solve the same workload issue. Here we present the challenges and needs of software engineers concerning performance testing in general and discuss how our contribution help address some of these challenges.

Software engineers design their systems with performance in mind regardless of the requirements. For example, in file management applications, the performance requirements might not be formally stated. However, there is a general understanding that it is not acceptable for a file creation to take minutes or even seconds. Software engineers usually rely on standard software architecture tactics (Bass, Clements, & Kazman, 2012) or even on the experience of the developers to meet the performance constraints.

Moreover, developers make decisions early on the application development cycle about software performance. A design might include hardware or software solutions, depending on solutions that involve hardware are usually limited to financial constraints. The hardware solutions have their limitations and often are applicable if the requirements are guaranteed to exceed individual hardware capabilities (Ammons et al., 1997; Ohmann et al., 2014). For example, a web-based application that processes tens of millions of requests in a few seconds might

replicate the software on multiple machines to handle more requests on time. However, many performance issues are fixable within the software design. Some performance issues are not solvable even if replicated over many pieces of hardware.

How to decide on what is acceptable as performance is another factor software engineers need to tackle. Most frequently, it is clear what would be the satisfactory execution or response time. However, as applications become functionally complete, previous measures might not hold. For example, developers need to anticipate the user's short-term memory in completing a task (Molyneaux, 2009). If the cumulative time to complete a task using the system exceeds a threshold of the user's short-term memory, then the application's usability degrades. Thus, previously identified acceptable response time changes. Consequently, finding and fixing those tasks becomes harder.

A severe problem that could arise is the developer's mis-anticipation of workload. Most reported performance issues arise from an unanticipated workload or library use misunderstanding (Jin et al., 2012; Zaman, Adams, & Hassan, 2012). Regardless of the developer's experience, it is hard to know how users will use the application. Any unanticipated use of the system could create severe performance issues. Even harder is finding these performance issues on deployed applications.

Our contribution helps software engineers gain better performance understanding in a shorter time than established techniques (Lemieux et al., 2018; Petsios et al., 2017). The time needed to find an extreme workload usually takes less than 10 minutes. Any additional time is usually only needed to refine the workload pattern. Although fuzzing techniques would eventually reach the same workload pattern, they usually require significantly more time than our approach.

Moreover, although the grammar use as a base in our case can be seen as a barrier, it actually can be advantageous. Fuzzing techniques assume the absence of any input structure and try all possible input by mutating seeds at the byte level. Unlike security, performance testing tends to be conducted within known possible inputs. Furthermore, a developer could test some modules of the application in isolation from the whole system. Using grammars allows the software engineer to have different performance testing abstractions by providing particular grammar for targeted modules.

6.2 Conclusion

This dissertation identified generating pathological input as a problem in the software performance analysis field. We provided background on established performance analysis techniques highlighting the trade-offs of each and how the research evolved within the domain. We then explained closely related techniques that use different strategies (e.g., machine learning, genetic algorithms, and fuzzing) to generate pathological input.

We identified possible research opportunities based on the limitation we identified on the state-of-the-art pathological input generator (PerfFuzz). We mainly explored using interactive search techniques such as reinforcrement learning (RL) and Monte Carlo tree search (MCTS) to find effective functions for generating pathological inputs of different real-world applications using easy-to-obtain input models. We used context-free grammar (CFG) as a base for generating input and formalized the CFG environment within an input generator that ensures bounding the generated input size.

Our adaptation of RL overcomes different limitations of using RL to generate pathological inputs. Our technique, PerfRL, adapts a long short-term

memory (LSTM) model to maintain derivation context, memory of the steps leading to each grammar choice. Without the derivation context, the model's accuracy will deteriorate due to inherited context loss from CFGs. Moreover, we introduce different state function abstractions to allow for different features. We also adapt various reward methods to dynamically adapt to the growing cost of executing inputs as the agent explores the target application's environment. Our implementation of PerfRL shows that we can find pathological inputs for trivial applications much faster than the fuzzing techniques. However, we identified limitations that indicate we must tune the training model's hyper-parameters for each tested application. Thus, we lose generality.

Adaptation of MCTS in a tool we called TreeLine proved more fruitful. Similar to RL, adapting MCTS to find pathological inputs requires enhancement of the traditional MCTS algorithm. Most notably, we introduce an adaptive reward function based on quantile streams, use lazy search based on hot nodes to speed up the search process and avoid falling into local maxima, refresh the search tree based on a dynamic evaluation of exhausted trees, and use a simple bigram heavy rollout strategy to bias toward known expensive patterns.

Our evaluation of TreeLine focuses on comparing it to PerfFuzz (the state-of-the-art pathological input generator) over a small but diverse set of real-world applications. The results show that we outperform the fuzzing-based technique on structured inputs while maintaining a very close performance on applications with unstructured inputs. Specifically, we find inputs that are 7.4x as expensive as the input found by PerfFuzz on a graph layout application (`graphviz`) and 2.7x on a lexical analyzer (`flex`) for which we immediately exercise a timeout (i.e., the maximum possible cost for the given grammar). While for

applications with unstructured input such as `wf`and `libxml2`, PerfFuzz finds inputs that are 1.01x and 1.1x respectively as expensive as the ones found by TreeLine. In our tests, TreeLine found pathological input in less than 10 minutes, while fuzzing techniques may require hours to find the same pathological inputs. Additionally, we showed how TreeLine can scale to larger input bounds (budget) while still finding similar expensive patterns.

6.3 Future Work

In addition to apparent technique-based enhancements such as expanding the evaluation to more target applications and studying the applicability of using different grammar synthesizers, we define several new directions for future work.

6.3.1 Branch-Based Search Tree. Our work in TreeLine and PerfRL uses a symbol-based search tree to generate pathological inputs. The search tree design poses a challenge in correcting earlier decisions made in the search tree. Focusing on TreeLine as an example, the earlier few decisions made in the search tree compose a derivation that can significantly affect the search quality. As the search moves deeper into the tree, the search thoroughness becomes better at the bottom of the tree. Lower nodes in the tree are expected to get more exploration opportunities. However, any very early wrong decision (i.e., at the top of the tree) requires a thorough exploration of the whole branch before it can be changed. These cases were apparent in applications such as the word frequency counter. And the larger the budget is given for input; the more severe this problem would become.

Adapting a branch-based swapping for known good branches could help mitigate the severity of the problem. For example, we can track and rank good low branches to be used in future established trees instead of repeating the same

exploration at the bottom of each tree. Thus, with each tree refresh, we explore the top of the tree more often than earlier ones.

6.3.2 Intelligent Heavy Rollouts. The heavy rollout we implemented is a straightforward enhancement of tracking pair symbol weights. Moreover, the reinforcement learning application we presented is very complicated; making minor enhancements comes with a high cost in search (e.g., longer time). A possible merge between the Monte Carlo tree search approach and reinforcement learning could lead to significant enhancements on search quality.

Instead of using a neural network-based model for reinforcement learning, it is possible to use a tabular-based approach to drive the rollouts in the Monte Carlo tree search. For each production rule, we can establish a designated tabular reinforcement learning model. Consequently, for each random rollout, we consult the reinforcement learning model for each evaluated node. Training a tabular-based reinforcement learning model is much cheaper than a neural network-based model regardless of the number of models we have to train. Furthermore, it might offers a better learner than our simple bigram bias rollout.

6.3.3 Enhanced Heuristics. The tree refresh mechanism we introduced with TreeLine is a critical method that helps us reach expensive input faster. However, the decision to drop a tree and start with a fresh one is simple and might not generalize over new target applications. It essentially depends on a single factor (cost redundancy) to make the refresh decision. We mitigate the shortcoming of such single factor dependency with a dynamic threshold and number of samples to consider to generalize over different target applications. A richer heuristics that drive tree refresh decisions could introduce a more stable and accurate search method.

APPENDIX

COMPLETE GRAMMARS

This appendix provides a reference for the complete grammars used in testing each target application.

⟨*Entry*⟩ ::= ⟨*Entry*⟩ ⟨*Word*⟩ | ⟨*Entry*⟩ ⟨*Break*⟩ | ⟨*MPT*⟩ ;

⟨*Word*⟩ ::= ⟨*Char*⟩ | ⟨*Digit*⟩ ;

⟨*Break*⟩ ::= ' ' | '\n' ;

⟨*Char*⟩ ::= 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J'
| 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S'
| 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | 'a' | 'b'
| 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l'
| 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v'
| 'w' | 'x' | 'y' | 'z' ;

⟨*Digit*⟩ ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;

⟨*MPT*⟩ ::= λ ;

Grammar 5. Manually written grammar for `wf`.

⟨*entry*⟩ ::= ⟨*entry*⟩ ⟨*char*⟩ | ⟨*MPT*⟩ ;

⟨*char*⟩ ::= '<' | '>' | ⟨*Alphanumeric*⟩ | ⟨*SpecialCharacters*⟩ ;

⟨*Alphanumeric*⟩ ::= '1' | '2' | '3' | 'a' | 'b' | 'c' ;

⟨*SpecialCharacters*⟩ ::= '-' | '.' | '/' | ':' | '?' | '_' ;

⟨*MPT*⟩ ::= λ ;

Grammar 6. Manually written grammar for `libxml2`.

⟨*graph*⟩ ::= ⟨*hdr*⟩ ⟨*body*⟩ ;

⟨*hdr*⟩ ::= ⟨*optstrict*⟩ 'digraph' ⟨*optgraphname*⟩ ;

⟨*body*⟩ ::= '{' ⟨*optstmtlist*⟩ '}' ;

⟨*optstmtlist*⟩ ::= ⟨*stmtlist*⟩ | ⟨*MPT*⟩ ;

⟨*optstrict*⟩ ::= 'strict ' | ⟨*MPT*⟩ ;

⟨*optgraphname*⟩ ::= ' ' ⟨*atom*⟩ | ⟨*MPT*⟩ ;

⟨*atom*⟩ ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
| 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I'
| 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R'
| 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | '_'
| 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j'
| 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't'
| 'u' | 'v' | 'w' | 'x' | 'y' | 'z' ;

⟨*stmtlist*⟩ ::= ⟨*stmtlist*⟩ ⟨*stmt*⟩ | ⟨*stmt*⟩ ;

⟨*stmt*⟩ ::= ⟨*attrstmt*⟩ ⟨*optsemi*⟩ | ⟨*compound*⟩ ⟨*optsemi*⟩ ;

⟨*attrstmt*⟩ ::= ⟨*attrtype*⟩ ⟨*optmacroname*⟩ ⟨*attrlist*⟩ | ⟨*attrassignment*⟩
;

⟨*optsemi*⟩ ::= ';' | ⟨*MPT*⟩ ;

⟨*compound*⟩ ::= ⟨*simple*⟩ ⟨*rcompound*⟩ ⟨*optattr*⟩ ;

⟨*simple*⟩ ::= ⟨*nodelist*⟩ | ⟨*subgraph*⟩ ;

⟨*rcompound*⟩ ::= ⟨*edgeop*⟩ ⟨*simple*⟩ ⟨*rcompound*⟩ | ⟨*MPT*⟩ ;

⟨*optattr*⟩ ::= ⟨*attrlist*⟩ | ⟨*MPT*⟩ ;

⟨*nodelist*⟩ ::= ⟨*node*⟩ | ⟨*nodelist*⟩ ',' ⟨*node*⟩ ;

⟨*subgraph*⟩ ::= ⟨*optsubghdr*⟩ ⟨*body*⟩ ;

⟨*edgeop*⟩ ::= '->' | ⟨*MPT*⟩ ;

⟨*node*⟩ ::= ⟨*atom*⟩ | ⟨*atom*⟩ ':' ⟨*port*⟩ ;

⟨*port*⟩ ::= 'n' | 'ne' | 'e' | 'se' | 's' | 'sw' | 'w' | 'nw' | 'c' | '_'
;

⟨*attrtype*⟩ ::= 'graph' | 'node' | 'edge' ;

⟨*optmacroname*⟩ ::= ' ' ⟨*atom*⟩ '=' | ⟨*MPT*⟩ ;

⟨*attrlist*⟩ ::= ⟨*optattr*⟩ '[' ⟨*optattrdefs*⟩ ']' ;

⟨*optattrdefs*⟩ ::= ⟨*optattrdefs*⟩ ⟨*attrdefs*⟩ | ⟨*MPT*⟩ ;

⟨*attrdefs*⟩ ::= ⟨*attritem*⟩ ⟨*optseparator*⟩ ;

⟨*attritem*⟩ ::= ⟨*attrassignment*⟩ | ⟨*attrmacro*⟩ ;

⟨*optseparator*⟩ ::= ';' | ',' | ⟨*MPT*⟩ ;

⟨*attrassignment*⟩ ::= ⟨*atom*⟩ '=' ⟨*atom*⟩ ;

⟨*attrmacro*⟩ ::= '@' ⟨*atom*⟩ ;

⟨*optsubghdr*⟩ ::= 'subgraph ' ⟨*atom*⟩ | 'subgraph' | ⟨*MPT*⟩ ;

⟨*MPT*⟩ ::= λ ;

Grammar 7. Parser-based grammar for `graphviz`.

⟨*flexspec*⟩ ::= ⟨*definitions*⟩ '%%\n' ⟨*rules*⟩ '%%\n' ;

⟨*definitions*⟩ ::= λ ;

⟨*rules*⟩ ::= {⟨*regex*⟩ ⟨*sp*⟩ ';\n'} ;

⟨*macro*⟩ ::= ⟨*mname*⟩ ⟨*sp*⟩ ⟨*regex*⟩ '\n' ;

⟨*mname*⟩ ::= [A-Z] {[A-Z]} ;

⟨*sp*⟩ ::= ' ' ;

⟨*regex*⟩ ::= ⟨*r_term*⟩ ⟨*optContext*⟩ ;

⟨*r_term*⟩ ::= ⟨*normal_char*⟩ | '.' | ⟨*char_class*⟩ | ⟨*kleene*⟩
| ⟨*kleene_plus*⟩ | ⟨*optional*⟩ | ⟨*repeat_range*⟩ | ⟨*quoted*⟩
| ⟨*escape_seq*⟩ | ⟨*group*⟩ | ⟨*seq*⟩ | ⟨*either*⟩ ;

⟨*optContext*⟩ ::= '/' ⟨*r_term*⟩ | λ ;

⟨*normal_char*⟩ ::= [a-zA-Z0-9] | ⟨*escape_seq*⟩ ;

⟨*char_class*⟩ ::= '[' '^' | λ ⟨*range*⟩ {⟨*range*⟩} ']' ;

⟨*kleene*⟩ ::= ⟨*r_term*⟩ '*' ;

⟨*kleene_plus*⟩ ::= ⟨*r_term*⟩ '+' ;

⟨*optional*⟩ ::= ⟨*r_term*⟩ '?' ;

⟨*repeat_range*⟩ ::= ⟨*r_term*⟩ '{' ⟨*how_many*⟩ '}' ;

⟨*quoted*⟩ ::= ["] ⟨*char*⟩ {⟨*char*⟩} ["] ;

⟨*escape_seq*⟩ ::= '\\' ⟨*escaped*⟩ ;

⟨*group*⟩ ::= '(' ⟨*r_term*⟩ ')' ;

⟨*seq*⟩ ::= ⟨*r_term*⟩ ⟨*r_term*⟩ ;

⟨*either*⟩ ::= ⟨*r_term*⟩ '| ' ⟨*r_term*⟩ ;

⟨*char*⟩ ::= [a-zA-Z0-9 ,*+?-] | ⟨*escape_seq*⟩ ;

⟨*range*⟩ ::= ⟨*char*⟩ {⟨*char*⟩} | ⟨*char*⟩ '-' ⟨*char*⟩ ;

⟨*escaped*⟩ ::= [a-zA-Z0-9] | [-+*?\\0.] | '[' | ']' | [0-9] [0-9] [0-9]
| 'x' [0-9a-f] [0-9a-fA-F] ;

⟨*how_many*⟩ ::= [0-9] {[0-9]} | [0-9] {[0-9]} ',' {[0-9]} ;

⟨*expansion*⟩ ::= ‘{’ ⟨*mname*⟩ ‘}’ ;

⟨*in_start_condition*⟩ ::= ‘<’ [a-zA-Z0-9] {[a-zA-Z0-9]} ‘>’ ⟨*r_term*⟩ ;

Grammar 8. Documentation-based grammar for `flex`.

REFERENCES CITED

Ahmad, T., Ashraf, A., Truscan, D., & Porres, I. (2019, Aug). Exploratory performance testing using reinforcement learning. In *2019 45th euromicro conference on software engineering and advanced applications (seaa)* (p. 156-163). doi: 10.1109/SEAA.2019.00032

Ammons, G., Ball, T., & Larus, J. R. (1997, May). Exploiting hardware performance counters with flow and context sensitive profiling. *ACM SIGPLAN Notices*, *32*(5), 85–96. Retrieved from `http://dx.doi.org/10.1145/258916.258924` doi: 10.1145/258916.258924

Aschermann, C., Frassetto, T., Holz, T., Jauernig, P., Sadeghi, A.-R., & Teuchert, D. (2019, 01). Nautilus: Fishing for deep bugs with grammars.. doi: 10.14722/ndss.2019.23412

AT&T Labs Research. (2021, Mar). *Graphviz - graph visualization software (version 2.47.0).* `https://gitlab.com/graphviz/graphviz/-/package_files/8183714/download`. (Accessed: Jul 2021)

Ayala-Rivera, V., Kaczmarski, M., Murphy, J., Darisa, A., & Portillo-Dominguez, A. O. (2018). One size does not fit all. *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering - ICPE '18*. Retrieved from `http://dx.doi.org/10.1145/3184407.3184418` doi: 10.1145/3184407.3184418

Baier, H., & Drake, P. D. (2010). The power of forgetting: Improving the last-good-reply policy in monte carlo go. *IEEE Transactions on Computational Intelligence and AI in Games*, *2*(4), 303-309. doi: 10.1109/TCIAIG.2010.2100396

Ball, T., & Larus, J. R. (1994, Jul). Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, *16*(4), 1319–1360. Retrieved from `http://dx.doi.org/10.1145/183432.183527` doi: 10.1145/183432.183527

Ball, T., & Larus, J. R. (1996). Efficient path profiling. In *Proceedings of the 29th annual acm/ieee international symposium on microarchitecture* (pp. 46–57). Washington, DC, USA: IEEE Computer Society. Retrieved from `http://dl.acm.org/citation.cfm?id=243846.243857`

Ball, T., Mataga, P., & Sagiv, M. (1998). Edge profiling versus path profiling. *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '98*. Retrieved from http://dx.doi.org/10.1145/268946.268958 doi: 10.1145/268946.268958

Balsamo, S., Di Marco, A., Inverardi, P., & Simeoni, M. (2004, May). Model-based performance prediction in software development: A survey. *IEEE Trans. Softw. Eng.*, *30*(5), 295–310. Retrieved from http://dx.doi.org/10.1109/TSE.2004.9 doi: 10.1109/TSE.2004.9

Bass, L., Clements, P., & Kazman, R. (2012). *Software architecture in practice* (3rd ed.). Addison-Wesley Professional.

Bastani, O., Sharma, R., Aiken, A., & Liang, P. (2017, Sep). Synthesizing program input grammars. *ACM SIGPLAN Notices*, *52*(6), 95–110. Retrieved from http://dx.doi.org/10.1145/3140587.3062349 doi: 10.1145/3140587.3062349

Bentley, J. L. (1982). *Writing efficient programs*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Bergel, A., Nierstrasz, O., Renggli, L., & Ressia, J. (2011). Domain-specific profiling. In J. Bishop & A. Vallecillo (Eds.), *Objects, models, components, patterns* (pp. 68–82). Berlin, Heidelberg: Springer Berlin Heidelberg.

Blackburn, S. M., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., ... et al. (2006). The dacapo benchmarks: java benchmarking development and analysis. *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications - OOPSLA '06*. Retrieved from http://dx.doi.org/10.1145/1167473.1167488 doi: 10.1145/1167473.1167488

Boehme, D., Gamblin, T., Beckingsale, D., Bremer, P., Gimenez, A., LeGendre, M., ... Schulz, M. (2016). Caliper: Performance introspection for hpc software stacks. In *Sc '16: Proceedings of the international conference for high performance computing, networking, storage and analysis* (pp. 550–560). doi: 10.1109/SC.2016.46

Brocanelli, M., & Wang, X. (2018). Hang doctor: runtime detection and diagnosis of soft hangs for smartphone apps. *Proceedings of the Thirteenth EuroSys Conference on - EuroSys '18*. Retrieved from http://dx.doi.org/10.1145/3190508.3190525 doi: 10.1145/3190508.3190525

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., ... Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, *4*(1), 1-43. doi: 10.1109/TCIAIG.2012.2186810

Brünink, M., & Rosenblum, D. S. (2016). Mining performance specifications. In *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering* (pp. 39–49). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/2950290.2950314` doi: 10.1145/2950290.2950314

Burnim, J., Juvekar, S., & Sen, K. (2009). Wise: Automated test generation for worst-case complexity. *2009 IEEE 31st International Conference on Software Engineering*. Retrieved from `http://dx.doi.org/10.1109/ICSE.2009.5070545` doi: 10.1109/icse.2009.5070545

Cadar, C., Dunbar, D., & Engler, D. (2008). Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th usenix conference on operating systems design and implementation* (pp. 209–224). Berkeley, CA, USA: USENIX Association. Retrieved from `http://dl.acm.org/citation.cfm?id=1855741.1855756`

Chaslot, G., Bakkes, S., Szita, I., & Spronck, P. (2008). Monte-carlo tree search: A new framework for game ai. In *Proceedings of the fourth aaai conference on artificial intelligence and interactive digital entertainment* (pp. 216–217). AAAI Press.

Chen, B., Liu, Y., & Le, W. (2016). Generating performance distributions via probabilistic symbolic execution. In *Proceedings of the 38th international conference on software engineering* (pp. 49–60). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/2884781.2884794` doi: 10.1145/2884781.2884794

Chen, Z., Chen, B., Xiao, L., Wang, X., Chen, L., Liu, Y., & Xu, B. (2018). Speedoo: prioritizing performance optimization opportunities. *Proceedings of the 40th International Conference on Software Engineering - ICSE '18*. Retrieved from `http://dx.doi.org/10.1145/3180155.3180229` doi: 10.1145/3180155.3180229

Coppa, E., Demetrescu, C., & Finocchi, I. (2012). Input-sensitive profiling. In *Proceedings of the 33rd acm sigplan conference on programming language design and implementation* (pp. 89–98). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/2254064.2254076` doi: 10.1145/2254064.2254076

Coulom, R. (2006). Efficient selectivity and backup operators in monte-carlo tree search. In *Proceedings of the 5th international conference on computers and games* (pp. 72–83). Berlin, Heidelberg: Springer-Verlag.

Curtsinger, C., & Berger, E. D. (2015). Coz: Finding code that counts with causal profiling. *Proceedings of the 25th Symposium on Operating Systems Principles - SOSP '15*. Retrieved from `http://dx.doi.org/10.1145/2815400.2815409` doi: 10.1145/2815400.2815409

de Barros, M. (2021, Apr). *wf - simple word frequency counter (version 0.41).* `http://ftp.altlinux.org/pub/distributions/ALTLinux/Sisyphus/x86_64/SRPMS.classic/wf-0.41-alt2.src.rpm`. (Accessed: Apr 2021)

D'Elia, D. C., & Demetrescu, C. (2013). Ball-larus path profiling across multiple loop iterations. *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications - OOPSLA'13*. Retrieved from `http://dx.doi.org/10.1145/2509136.2509521` doi: 10.1145/2509136.2509521

Della Toffola, L., Pradel, M., & Gross, T. R. (2015). Performance problems you can fix: a dynamic analysis of memoization opportunities. *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2015*. Retrieved from `http://dx.doi.org/10.1145/2814270.2814290` doi: 10.1145/2814270.2814290

Dhok, M., & Ramanathan, M. K. (2016). Directed test generation to detect loop inefficiencies. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*. Retrieved from `http://dx.doi.org/10.1145/2950290.2950360` doi: 10.1145/2950290.2950360

Duesterwald, E., & Bala, V. (2000, November). Software profiling for hot path prediction: Less is more. *SIGARCH Comput. Archit. News*, *28*(5), 202–211. Retrieved from `http://doi.acm.org/10.1145/378995.379241` doi: 10.1145/378995.379241

Dugan, R. F. (2004). Performance lies my professor told me. *Proceedings of the fourth international workshop on Software and performance - WOSP '04*. Retrieved from `http://dx.doi.org/10.1145/974044.974050` doi: 10.1145/974044.974050

Dunning, T. (2021). The t-digest: Efficient estimates of distributions. *Software Impacts*, *7*, 100049. Retrieved from https://www.sciencedirect.com/science/article/pii/S2665963820300403 doi: https://doi.org/10.1016/j.simpa.2020.100049

Dunning, T., & Ertl, O. (2019). *Computing extremely accurate quantiles using t-digests.* Retrieved from https://arxiv.org/abs/1902.04023

GeeksForGeeks. (2021, Jan). *Quicksort.* https://www.geeksforgeeks.org/quick-sort/. (Accessed: Jan 2021)

Goldsmith, S. F., Aiken, A. S., & Wilkerson, D. S. (2007). Measuring empirical computational complexity. *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering - ESEC-FSE '07*. Retrieved from http://dx.doi.org/10.1145/1287624.1287681 doi: 10.1145/1287624.1287681

Gopinath, R., Mathis, B., & Zeller, A. (2020). Mining input grammars from dynamic control flow. In *Proceedings of the 28th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (pp. 172–183). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3368089.3409679 doi: 10.1145/3368089.3409679

Graham, S. L., Kessler, P. B., & Mckusick, M. K. (1982, June). Gprof: A call graph execution profiler. *SIGPLAN Not.*, *17*(6), 120–126. Retrieved from http://doi.acm.org/10.1145/872726.806987 doi: 10.1145/872726.806987

Graham, S. L., Kessler, P. B., & McKusick, M. K. (2004, Apr). gprof: a call graph execution profiler. *ACM SIGPLAN Notices*, *39*(4), 49. Retrieved from http://dx.doi.org/10.1145/989393.989401 doi: 10.1145/989393.989401

Grant, S., Cordy, J. R., & Skillicorn, D. (2008, Oct). Automated concept location using independent component analysis. *2008 15th Working Conference on Reverse Engineering*. Retrieved from http://dx.doi.org/10.1109/WCRE.2008.49 doi: 10.1109/wcre.2008.49

Grechanik, M., Fu, C., & Xie, Q. (2012). Automatically finding performance problems with feedback-directed learning software testing. In *Proceedings of the 34th international conference on software engineering* (pp. 156–166). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=2337223.2337242

Greibach, S. A. (1965, January). A new normal-form theorem for context-free phrase structure grammars. *J. ACM*, *12*(1), 42–52. Retrieved from https://doi.org/10.1145/321250.321254 doi: 10.1145/321250.321254

Guo, P. J., & Engler, D. (2011). Using automatic persistent memoization to facilitate data analysis scripting. *Proceedings of the 2011 International Symposium on Software Testing and Analysis - ISSTA '11*. Retrieved from http://dx.doi.org/10.1145/2001420.2001455 doi: 10.1145/2001420.2001455

Han, X., & Yu, T. (2016). An empirical study on performance bugs for highly configurable software systems. *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '16*. Retrieved from http://dx.doi.org/10.1145/2961111.2962602 doi: 10.1145/2961111.2962602

Harman, M., Mansouri, S. A., & Zhang, Y. (2012, December). Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, *45*(1). Retrieved from https://doi.org/10.1145/2379776.2379787 doi: 10.1145/2379776.2379787

Hellendoorn, V. J., & Devanbu, P. (2017). Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th joint meeting on foundations of software engineering* (pp. 763–773). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3106237.3106290 doi: 10.1145/3106237.3106290

Herodotou, H., & Babu, S. (2011, August). Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *Proc. VLDB Endow.*, *4*(11), 1111–1122. Retrieved from https://doi.org/10.14778/3402707.3402746 doi: 10.14778/3402707.3402746

Hoefler, T., Gropp, W., Kramer, W., & Snir, M. (2011). Performance modeling for systematic performance tuning. In *State of the practice reports* (pp. 6:1–6:12). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2063348.2063356 doi: 10.1145/2063348.2063356

Holler, C., Herzig, K., & Zeller, A. (2012). Fuzzing with code fragments. In *Proceedings of the 21st usenix conference on security symposium* (p. 38). USA: USENIX Association.

Hopcroft, J. E., & Ullman, J. D. (1969). *Formal languages and their relation to automata*. Addison-Wesley Longman Publishing Co., Inc.

Höschele, M., & Zeller, A. (2016). Mining input grammars from dynamic taints. In *2016 31st ieee/acm international conference on automated software engineering (ase)* (p. 720-725).

Infante, A. (2014). Identifying caching opportunities, effortlessly. *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014*. Retrieved from http://dx.doi.org/10.1145/2591062.2591198 doi: 10.1145/2591062.2591198

Jin, G., Song, L., Shi, X., Scherpelz, J., & Lu, S. (2012). Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd acm sigplan conference on programming language design and implementation* (pp. 77–88). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2254064.2254075 doi: 10.1145/2254064.2254075

Jsfuzz - coverage guided fuzz testing for javascript. (2019, Oct). https://github.com/fuzzitdev/jsfuzz. (Accessed: September 2021)

Kalchbrenner, N., Grefenstette, E., & Blunsom, P. (2014). A convolutional neural network for modelling sentences. *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Retrieved from http://dx.doi.org/10.3115/v1/P14-1062 doi: 10.3115/v1/p14-1062

Kang, Y., Zhou, Y., Xu, H., & Lyu, M. R. (2015). *Persisdroid: Android performance diagnosis via anatomizing asynchronous executions.*

Kang, Y., Zhou, Y., Xu, H., & Lyu, M. R. (2016). Diagdroid: Android performance diagnosis via anatomizing asynchronous executions. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*. Retrieved from http://dx.doi.org/10.1145/2950290.2950316 doi: 10.1145/2950290.2950316

Kim, C. H., Rhee, J., Lee, K. H., Zhang, X., & Xu, D. (2016). Perfguard: binary-centric application performance monitoring in production environments. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*. Retrieved from http://dx.doi.org/10.1145/2950290.2950347 doi: 10.1145/2950290.2950347

Kocsis, L., & Szepesvári, C. (2006). Bandit based monte-carlo planning. *IN: ECML-06. NUMBER 4212 IN LNCS*, 282–293. Retrieved from https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.102.1296 doi: 10.1.1.102.1296

Korel, B. (1990). Automated software test data generation. *IEEE Transactions on Software Engineering*, *16*(8), 870–879. doi: 10.1109/32.57624

Koziolek, H. (2010, August). Performance evaluation of component-based software systems: A survey. *Perform. Eval.*, *67*(8), 634–658. Retrieved from http://dx.doi.org/10.1016/j.peva.2009.07.007 doi: 10.1016/j.peva.2009.07.007

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017, May). Imagenet classification with deep convolutional neural networks. *Commun. ACM*, *60*(6), 84–90. Retrieved from https://doi.org/10.1145/3065386 doi: 10.1145/3065386

Kulkarni, N., Lemieux, C., & Sen, K. (2021). *Learning highly recursive input grammars.*

Küstner, T., Weidendorfer, J., & Weinzierl, T. (2010). Argument controlled profiling. In *Proceedings of the 2009 international conference on parallel processing* (pp. 177–184). Berlin, Heidelberg: Springer-Verlag. Retrieved from http://dl.acm.org/citation.cfm?id=1884795.1884819

Larus, J. R. (1999). Whole program paths. *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation - PLDI '99*. Retrieved from http://dx.doi.org/10.1145/301618.301678 doi: 10.1145/301618.301678

Lemieux, C., Padhye, R., Sen, K., & Song, D. (2018). Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th acm sigsoft international symposium on software testing and analysis* (pp. 254–265). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3213846.3213874 doi: 10.1145/3213846.3213874

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... Wierstra, D. (2015). *Continuous control with deep reinforcement learning.*

Liu, Y., Xu, C., & Cheung, S.-C. (2014). Characterizing and detecting performance bugs for smartphone applications. *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*. Retrieved from http://dx.doi.org/10.1145/2568225.2568229 doi: 10.1145/2568225.2568229

Luo, Q. (2016a). Automatic performance testing using input-sensitive profiling. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*. Retrieved from http://dx.doi.org/10.1145/2950290.2983975 doi: 10.1145/2950290.2983975

Luo, Q. (2016b). Input-sensitive performance testing. In *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering* (pp. 1085–1087). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2950290.2983953 doi: 10.1145/2950290.2983953

Luo, Q., Poshyvanyk, D., & Grechanik, M. (2016). Mining performance regression inducing code changes in evolving software. *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16*. Retrieved from http://dx.doi.org/10.1145/2901739.2901765 doi: 10.1145/2901739.2901765

Luo, Q., Poshyvanyk, D., Nair, A., & Grechanik, M. (2016). Forepost: a tool for detecting performance problems with feedback-driven learning software testing. *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16*. Retrieved from http://dx.doi.org/10.1145/2889160.2889164 doi: 10.1145/2889160.2889164

Mathis, B., Gopinath, R., Mera, M., Kampmann, A., Höschele, M., & Zeller, A. (2019). Parser-directed fuzzing. In *Proceedings of the 40th acm sigplan conference on programming language design and implementation* (pp. 548–560). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3314221.3314651 doi: 10.1145/3314221.3314651

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., ... Kavukcuoglu, K. (2016). *Asynchronous methods for deep reinforcement learning.*

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). *Playing atari with deep reinforcement learning.*

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... Hassabis, D. (2015, February). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533. Retrieved from http://dx.doi.org/10.1038/nature14236

Molyneaux, I. (2009). *The art of application performance testing: Help for programmers and quality assurance* (1st ed.). O'Reilly Media, Inc.

Mudduluru, R., & Ramanathan, M. K. (2016). Efficient flow profiling for detecting performance bugs. *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*. Retrieved from http://dx.doi.org/10.1145/2931037.2931066 doi: 10.1145/2931037.2931066

Mytkowicz, T., Diwan, A., Hauswirth, M., & Sweeney, P. F. (2010, May). Evaluating the accuracy of java profilers. *ACM SIGPLAN Notices*, *45*(6), 187. Retrieved from http://dx.doi.org/10.1145/1809028.1806618 doi: 10.1145/1809028.1806618

Nethercote, N., & Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation - PLDI '07*. Retrieved from http://dx.doi.org/10.1145/1250734.1250746 doi: 10.1145/1250734.1250746

Nguyen, K., & Xu, G. (2013). Cachetor: detecting cacheable data to remove bloat. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*. Retrieved from http://dx.doi.org/10.1145/2491411.2491416 doi: 10.1145/2491411.2491416

Nistor, A., Chang, P.-C., Radoi, C., & Lu, S. (2015). Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *Proceedings of the 37th international conference on software engineering - volume 1* (pp. 902–912). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=2818754.2818863

Nistor, A., Jiang, T., & Tan, L. (2013). Discovering, reporting, and fixing performance bugs. In *2013 10th working conference on mining software repositories (msr)* (pp. 237–246). doi: 10.1109/MSR.2013.6624035

Nistor, A., Song, L., Marinov, D., & Lu, S. (2013). Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 international conference on software engineering* (pp. 562–571). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=2486788.2486862

Ofenbeck, G., Steinmann, R., Caparros, V., Spampinato, D. G., & Püschel, M. (2014). Applying the roofline model. In *2014 ieee international symposium on performance analysis of systems and software (ispass)* (pp. 76–85). doi: 10.1109/ISPASS.2014.6844463

Ohmann, T., Herzberg, M., Fiss, S., Halbert, A., Palyart, M., Beschastnikh, I., & Brun, Y. (2014). Behavioral resource-aware model inference. In *Proceedings of the 29th acm/ieee international conference on automated software engineering* (pp. 19–30). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2642937.2642988 doi: 10.1145/2642937.2642988

Pacheco, C., & Ernst, M. D. (2007). Randoop: feedback-directed random testing for java. *Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion - OOPSLA '07*. Retrieved from http://dx.doi.org/10.1145/1297846.1297902 doi: 10.1145/1297846.1297902

Paxson, V. (2001, Feb). *Flex - a scanner generator - deficiencies.* https://ftp.gnu.org/old-gnu/Manuals/flex-2.5.4/html_node/flex_23.html. (Accessed: Aug 2021)

Paxson, V. (2017, May). *flex - the fast lexical analyzer - scanner generator for lexing in c and c++ (version 2.6.4).* https://github.com/westes/flex/archive/refs/tags/v2.6.4.tar.gz. (Accessed: Aug 2021)

pcre.org. (2019, 05). *Pcre - perl compatible regular expressions.* http://pcre.org/. Retrieved from http://pcre.org/

Perez, D., Samothrakis, S., & Lucas, S. (2014, Aug). Knowledge-based fast evolutionary mcts for general video game playing. In *2014 ieee conference on computational intelligence and games* (p. 1-8). doi: 10.1109/CIG.2014.6932868

Petsios, T., Zhao, J., Keromytis, A. D., & Jana, S. (2017). Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 acm sigsac conference on computer and communications security* (pp. 2155–2168). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3133956.3134073 doi: 10.1145/3133956.3134073

Porres, I., Ahmad, T., Rexha, H., Lafond, S., & Truscan, D. (2020). Automatic exploratory performance testing using a discriminator neural network. In *2020 ieee international conference on software testing, verification and validation workshops (icstw)* (p. 105-113). doi: 10.1109/ICSTW50294.2020.00030

Păsăreanu, C. S., Visser, W., Bushnell, D., Geldenhuys, J., Mehlitz, P., & Rungta, N. (2013, Feb). Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Automated Software Engineering*, *20*(3), 391–425. Retrieved from http://dx.doi.org/10.1007/s10515-013-0122-2 doi: 10.1007/s10515-013-0122-2

Sánchez, A. B., Delgado-Pérez, P., Medina-Bulo, I., & Segura, S. (2018). Search-based mutation testing to improve performance tests. *Proceedings of the Genetic and Evolutionary Computation Conference Companion on - GECCO '18*. Retrieved from http://dx.doi.org/10.1145/3205651.3205670 doi: 10.1145/3205651.3205670

Sandoval Alcocer, J. P., Bergel, A., & Valente, M. T. (2016). Learning from source code history to identify performance failures. *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering - ICPE '16*. Retrieved from http://dx.doi.org/10.1145/2851553.2851571 doi: 10.1145/2851553.2851571

Schadd, M. P. D., Winands, M. H. M., van den Herik, H. J., Chaslot, G. M. J. B., & Uiterwijk, J. W. H. M. (2008). Single-player monte-carlo tree search. In H. J. van den Herik, X. Xu, Z. Ma, & M. H. M. Winands (Eds.), *Computers and games* (pp. 1–12). Berlin, Heidelberg: Springer Berlin Heidelberg.

Selakovic, M., Glaser, T., & Pradel, M. (2017). An actionable performance profiler for optimizing the order of evaluations. *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2017*. Retrieved from http://dx.doi.org/10.1145/3092703.3092716 doi: 10.1145/3092703.3092716

Shen, D., Luo, Q., Poshyvanyk, D., & Grechanik, M. (2015). Automating performance bottleneck detection using search-based application profiling. In *Proceedings of the 2015 international symposium on software testing and analysis* (pp. 270–281). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2771783.2771816 doi: 10.1145/2771783.2771816

Shende, S. S., & Malony, A. D. (2006, May). The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, *20*(2), 287–311. Retrieved from http://dx.doi.org/10.1177/1094342006064482 doi: 10.1177/1094342006064482

Siegmund, N., Kolesnikov, S. S., Kästner, C., Apel, S., Batory, D., Rosenmüller, M., & Saake, G. (2012). Predicting performance via automated feature-interaction detection. In *Proceedings of the 34th international conference on software engineering* (pp. 167–177). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=2337223.2337243

Song, L., & Lu, S. (2017, May). Performance diagnosis for inefficient loops. *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. Retrieved from http://dx.doi.org/10.1109/ICSE.2017.41 doi: 10.1109/icse.2017.41

Srivastava, P., & Payer, M. (2021). Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th acm sigsoft international symposium on software testing and analysis* (pp. 244–256). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3460319.3464814 doi: 10.1145/3460319.3464814

Sumner, W. N., Zheng, Y., Weeratunge, D., & Zhang, X. (2010). Precise calling context encoding. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*. Retrieved from http://dx.doi.org/10.1145/1806799.1806875 doi: 10.1145/1806799.1806875

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. Cambridge, MA, USA: A Bradford Book.

Tikir, M. M., & Hollingsworth, J. K. (2002). Efficient instrumentation for code coverage testing. *Proceedings of the international symposium on Software testing and analysis - ISSTA '02*. Retrieved from http://dx.doi.org/10.1145/566172.566186 doi: 10.1145/566172.566186

van Hasselt, H., Guez, A., & Silver, D. (2015). *Deep reinforcement learning with double q-learning*.

Veillard, D. (2017, Oct). *Libxml2 - the xml c parser and toolkit of gnome (version 2.9.7)*. http://xmlsoft.org/sources/libxml2-2.9.7-rc1.tar.gz. (Accessed: June 2021)

Visser, W., Păsăreanu, C. S., & Khurshid, S. (2004). Test input generation with java pathfinder. *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis - ISSTA '04*. Retrieved from http://dx.doi.org/10.1145/1007512.1007526 doi: 10.1145/1007512.1007526

Vyukov, D. (2021, September). *go-fuzz: Randomized testing for go.* https://github.com/dvyukov/go-fuzz. (Accessed: September 2021)

Wert, A., Happe, J., & Happe, L. (2013). Supporting swift reaction: Automatically uncovering performance problems by systematic experiments. In *Proceedings of the 2013 international conference on software engineering* (pp. 552–561). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=2486788.2486861

Wu, Z., Johnson, E., Yang, W., Bastani, O., Song, D., Peng, J., & Xie, T. (2019). Reinam: Reinforcement learning for input-grammar inference. In *Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (pp. 488–498). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3338906.3338958 doi: 10.1145/3338906.3338958

Xiao, X., Han, S., Zhang, D., & Xie, T. (2013). Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. *Proceedings of the 2013 International Symposium on Software Testing and Analysis - ISSTA 2013*. Retrieved from http://dx.doi.org/10.1145/2483760.2483784 doi: 10.1145/2483760.2483784

Xu, G. (2012). Finding reusable data structures. *Proceedings of the ACM international conference on Object oriented programming systems languages and applications - OOPSLA '12*. Retrieved from http://dx.doi.org/10.1145/2384616.2384690 doi: 10.1145/2384616.2384690

Zalewski, M. (2013, 12). *American fuzzy lop.* https://lcamtuf.coredump.cx/afl/. (Accessed: 2020-12-26)

Zaman, S., Adams, B., & Hassan, A. E. (2012). A qualitative study on performance bugs. In *Proceedings of the 9th ieee working conference on mining software repositories* (pp. 199–208). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=2664446.2664477

Zaparanuks, D., & Hauswirth, M. (2012). Algorithmic profiling. In *Proceedings of the 33rd acm sigplan conference on programming language design and implementation* (pp. 67–76). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2254064.2254074 doi: 10.1145/2254064.2254074

Zhang, P., Elbaum, S., & Dwyer, M. B. (2011, Nov). Automatic generation of load tests. *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. Retrieved from http://dx.doi.org/10.1109/ASE.2011.6100093 doi: 10.1109/ase.2011.6100093

Zhang, S., & Ernst, M. D. (2014). Which configuration option should i change? In *Proceedings of the 36th international conference on software engineering* (pp. 152–163). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2568225.2568251 doi: 10.1145/2568225.2568251

Zhuang, X., Serrano, M. J., Cain, H. W., & Choi, J.-D. (2006). Accurate, efficient, and adaptive calling context profiling. *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation - PLDI '06*. Retrieved from http://dx.doi.org/10.1145/1133981.1134012 doi: 10.1145/1133981.1134012