Applying Statistical Language Modeling to Genetic Programming

Hodgdon Chase Stevens

Master of Science
Artificial Intelligence
School of Informatics
University of Edinburgh
2016

Abstract

Recent work has demonstrated the successful application of probabilistic techniques, as typically employed in natural language modeling, to formal languages, spurred by the increasing availability of large, publicly-accessible source code repositories. This dissertation presents the application of a language model of the Python programming language to genetic programming, a form of evolutionary algorithm in which potential solutions to a problem are represented as programs. To accomplish this, a new, multi-megatoken corpus of Python source code is compiled from a selection of code repositories; a statistical language model of Python is induced from this corpus, using — to the best of the author's knowledge — a novel application of natural language processing techniques to a formal language, so as to allow application of the language model to genetic programming. The incorporation of the language model in informing the search and selection processes of a genetic programming system, as applied to the task of automated program repair, is observed to result in higher-fitness individuals when compared to a baseline approach.

Acknowledgements

I would like to extend my most sincere gratitude to my supervisor, Dr. Ian Stark, for his valuable advice and guidance, to my parents, Richard and Catherine Stevens, for their continued and unconditional support of my studies, and to the University of Edinburgh, for the provision of computational resources necessary for this project.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Hodgdon Chase Stevens*)

Table of Contents

Chapter 1

Introduction

This chapter serves as an introduction to and overview of the dissertation, by touching upon the motivation behind the dissertation, what the dissertation seeks to accomplish, the work performed in service of the dissertation, and the results obtained from this work. Furthermore, an executive summary of the contents and structure of the accompanying chapters of the dissertation is provided.

1.1 Motivation

Genetic programming is an iterative optimization method from the class of search techniques known as "evolutionary algorithms". In these algorithms, successive populations of individuals, representing solutions to a given objective function, are mutated and recombined, over a number of generations; in the case of genetic programming, these individuals are tree-like structures, most commonly comprising executable programs [1].

Unlike many other evolutionary algorithms, such as typical formulations of genetic algorithms and differential evolution [2], genetic programming searches through a potentially infinite space of possible solutions. Because of this, it is of the utmost importance that measures are put in place to constrain and direct the search process, more so than with other search techniques. As reported in the literature, one method of accomplishing this has been the gradual induction of a probabilistic grammar during the search process, as informed by the relative fitnesses of each individual and the constructions from which they are composed [1, 3, 4, 5]. However, for this grammar to inform program search, it must first be learned over a number of generations — and, even then, is restricted to the constructions observed within the population.

Much work in the recent literature has been devoted to the statistical modeling of formal languages using techniques from the field of natural language processing [6, 7, 8], as spurred by increasing access to large corpora of source code. This invites the question: can these methods be used as an *a priori* means of better directing genetic programming search? In leveraging this work, in addition to other work from the natural language processing literature, a sufficiently expressive model of a programming language, as learned from a corpus, might be engineered to provide a means of promoting fruitful exploration of the genetic programming search space, while also limiting the generation of low-fitness or invalid solutions in the *initial* population — before any fitness evaluation is performed.

With particular reference to the work presented in Weimer et al. [9], Forrest et al. [10], and Le Goues et al. [11], the task of automated program repair via genetic programming appears quite amenable to the use of a statistical language model of a programming language. As software spends the majority of its lifecycle undergoing maintenance rather than new development, and as software maintenance constitutes nearly half of software engineering time in industry on average [12], an advance in the efficacy of automated repair methods, especially methods which might easily be adapted to a specific domain or codebase, could have broad impact.

1.2 Aim and objectives

The aim of the project presented in this dissertation is to determine the feasibility and possible benefits of incorporating statistical language models into genetic programming systems. An answer to this question has been sought by evaluating a genetic programming system's performance on the task of automated program repair, both with and without the use of a probabilistic language model — as learned from a multi-megatoken source code corpus — to inform genetic programming search. In order to accomplish this, the following objectives were established:

1. to compile a sufficiently large corpus of Python source code,
2. to select and apply an appropriate language modeling approach to the corpus,
3. to evaluate the efficacy of the approach chosen against a baseline measure,
4. to implement a genetic programming system incorporating the language model, and, finally,

5. to apply the genetic programming system to a suitable task, evaluating it against a baseline approach.

1.3 Results

The results of the application of a language-model-imbued genetic programming system to program repair seem to indicate an overall improvement in final population fitnesses — with fitness increasing in four out of ten test cases — as opposed to the performance observed when using a baseline, uniform-production-probability approach, which was only able to improve fitness in a single instance. Additionally, the best of the language models learned for use with the genetic programming system is shown to improve upon the aforementioned uniformly-smoothed language modeling approach by several orders of magnitude, as measured via estimated cross-entropy when parsing a test set of Python modules.

1.4 Organization of this dissertation

The remainder of this dissertation is divided into the following chapters:

- Chapter 2 provides an introduction to the fields of genetic programming and language modeling, with the intent of situating the content of this dissertation within the existing literature.
- Chapter 3 details the methods used in this dissertation and the design decisions made in corpus compilation, language modeling, genetic programming system implementation, and system evaluation, with special attention paid to the relationship of these decisions to previous work.
- Chapter 4 evaluates and discusses the results of the work undertaken, compares these with previous results, and provides suggestions for future work.
- Chapter 5 summarizes and concludes the dissertation.

Chapter 2

Background

In order to provide an introduction to the topics of language modeling and genetic programming, an overview and brief survey of the literature, as it relates to this work, is presented for both in this chapter.

2.1 Language modeling

2.1.1 Overview

Given an arbitrary sequence of tokens $w_1, ..., w_n$, a statistical language model defines a probability for the sequence $P(w_1, ..., w_n)$ with respect to some language L on which the model was trained [13]. Language models vary in their complexity and the extent to which they capture structural elements of language; for example, the unigram model, one of the simplest possible approaches, models sequences as unstructured, conditionally independent collections of tokens [14] wherein:

$$P(w_1, ..., w_n) = \prod_{i=1}^{n} P(w_i) \tag{2.1}$$

n-gram models, of which unigram models are a special case ($n = 1$), can capture the sequential nature of a string of tokens by modeling each token w_i as conditionally dependent on the preceding $n-1$ tokens $w_{i-1}, ..., w_{i+1-n}$ [13]. For example, in a trigram model (for which $n = 3$), given a start symbol ε [14]:

$$P(w_1, ..., w_n) = P(w_1|\varepsilon)P(w_2|w_1)\prod_{i=3}^{n} P(w_i|w_{i-1}, w_{i-2}) \tag{2.2}$$

Probabilistic context-free grammars (PCFGs), another form of language model, assume a latent tree structure underlying token sequences. PCFGs extend context-free grammars, which are defined by a set of non-terminal symbols N, a set of terminal

symbols Σ, a set of production rules R, and a choice of start symbol $S \in N$, with a set of probabilities over rules P. Every production rule in a PCFG describes the production of a sequence of one or more symbols $\beta \in (\Sigma \bigcup N)$ from a nonterminal "left-hand side" $A \in \Sigma$ with some probability $P(A \rightarrow \beta)$ [13]. For illustratory purposes, a worked example of sequence probabilities under a PCFG model is given in Appendix A.

Typically, maximum likelihood estimation is used in order to learn a language model from a corpus, in which probabilities are induced via normalized corpus frequencies. One well-known issue with this method is the question of how to handle unseen productions during model evaluation; while the assumption can be made that any productions not specifically observed in the training data should be assigned a probability of 0, in the aforementioned models, this "poisons the well" by inevitably resulting in an overall probability of 0 for the entirety of the sequence in which the unknown production was encountered [13]. To compensate for the possibility of unknown productions, therefore, a number of smoothing techniques have been developed. Perhaps the simplest is Laplace smoothing, in which, after training, the frequencies of each production observed are incremented by 1, with the "unknown" production thereby receiving a total frequency of 1; this method constitutes a special case of additive smoothing (also known as "add-α smoothing"), in which the quantity added can be any real-valued number [15].

2.1.2 Application to formal languages

The application of language modeling techniques to formal, artificial languages (as opposed to natural languages) is a relatively recent area of interest in the literature, with the majority of work to date focusing on the use of n-gram models. The seminal work presented in Hindle et al. [16] developed a number of n-gram models of the Java and C programming languages, using 15 and 24 megatoken corpora, respectively, and employed these models for the purpose of token suggestion, having achieved a cross-entropy (in Java) of 6 bits per token; Allamanis and Sutton [6] improved upon these results by reporting a cross-entropy measure of 4.9 bits per token using a trigram model trained on a gigatoken Java corpus. Tu et al. [7] reported a further order-of-magnitude reduction in cross-entropy on the same Java modeling task via the use of a cached n-gram model, wherein a priming effect was used to increase n-gram probability after recent observation during the course of incremental parsing. Additional work within the field of code attribution [8] has applied language modeling not to pro-

gramming languages, but to the individual coding styles of developers — employing a model consisting of learned features over abstract syntax trees (ASTs) in order to enable classification.

2.2 Genetic programming

2.2.1 Overview

Genetic programming, first described in Koza [17] and further developed in Koza [18], is a type of evolutionary algorithm, which themselves are a form of optimization algorithm [1, 2]. Broadly speaking, optimization algorithms seek to optimize[1] an objective function f by searching through a given space of possible solutions, potentially subject to one or more constraints. In evolutionary algorithms, this search is performed by iterating through populations of solutions, with the individuals comprising each previous population and the fitnesses thereof (as measured by f) being used to govern the creation of each successive generation; under the evolutionary algorithm paradigm, search is conceptualized as Darwinian evolution — the fitness of an individual (oftentimes spoken of as a "genome") within the population determines its likelihood for "selection", and evolutionary algorithms are said to "mutate", "breed", and produce "offspring" from selected individuals in order to arrive at the next set of search targets [2].

In genetic programming, individual solutions are tree structures, which (typically) represent executable programs. Most commonly, these trees are composed of terminal nodes representing constant values or program inputs, and nonterminal function nodes which act upon these terminal values or upon the outputs of other functions [1]; however, other tree structures may be employed, providing they can be executed and, hence, evaluated with regard to f — as is the case in this dissertation, in which Python ASTs are used.

Under the most basic formulation of genetic programming, an initial population is generated randomly from the set of available terminal and nonterminal nodes [1, 2]. Individuals in subsequent generations are created through the random replacement of nodes or subtrees in the fittest individuals ("mutation"), through the replacement of a subtree of one individual with that of another ("crossover"), or through a combination

[1] This is to say, to find either a global maximum or global minimum of the fitness landscape defined by f, as appropriate to the problem at hand.

of the two methods [2]. In this way, exploration of the neighborhood of the fittest solutions of the parent generation is achieved.

2.2.2 Challenges in comparison to other optimization algorithms

Whereas optimization methods such as genetic algorithms, particle swarm optimization, differential evolution, and estimation of distribution algorithms normally use fixed-sized vector representations for solutions, and thereby seek to find extrema within finite search spaces [2, 5], genetic programming generally operates on unbounded tree structures, resulting in an infinite search space; this, naturally, poses challenges which are not faced by other optimization methods, and has spurred the development of a number of techniques for directing and constraining genetic programming search [1, 5].

One such technique is strongly-typed genetic programming. In strongly-typed genetic programming, constraints on parent-child node relationships and tree structure are imposed, so as to avoid the evolution and subsequent evaluation of individuals containing known bad structures, with these constraints often taking the form of arity and type annotations [19]. For example, a set of constraints on a nonterminal implementing multiplication might be that exactly two children are specified, both of which are known to be of a numeric type, and that the parent of the nonterminal is known to accept a numeric type as an input value. As can be imagined, significant shortcomings of the strongly-typed genetic programming approach are that, normally, constraints must be manually specified before search is begun, and that constraints do not adapt to issues discovered at run-time.

To overcome these shortcomings, a number of more dynamic methods have been proposed whereby a probabilistic grammar over node combinations is gradually learned. In order to do so, an individual's fitness is used to estimate the relative correlated fitness impact of each of the node relationships occurring in that individual, with the constructions used most often in the most meritorious solutions becoming gradually more probable [1]. This approach allows for the entirely unsupervised learning of a grammar [20], but has also seen widespread usage in concert with strongly-typed methodologies [3, 21, 22]. In an interesting variant, Keber and Schuster [4] proposes the use of ant colony optimization (ACO) — an iterative, biologically-inspired, autocatalytic optimization method — to learn weights over an implicit grammar. This is done by representing programs as paths through a graph of all possible programs,

Figure 2.1: Comparison of two trees representing equivalent expressions in propositional logic: a parsimonious version $P \wedge Q$ (left) and a bloated version $(P \wedge Q) \vee (\bot \wedge \bot)$ (right). Consider a mutation scenario in which one of the terminal nodes, selected at random, is to be replaced with a new terminal symbol R. Whereas in the parsimonious tree, this mutation will be guaranteed to result in an expression which is not equivalent to the original, in the bloated tree, the chance of this occurring is reduced to only 50%.

with the selection probability of each edge within the graph being dependent on the fitnesses of the individuals in which that edge is present, and evolving over a number of generations.

A weakness of all the aforementioned probabilistic grammar methods is that probabilities must be learned solely during the search process, and are typically initialized such that all productions are given a uniform probability — an important distinction between previous work and the methods presented in this dissertation.

2.2.3 Bloat

In addition to the challenges imposed by the tendency toward extremely large or infinite search spaces in genetic programming, a great deal of the genetic programming literature is devoted to the problematic phenomenon known as "bloat", wherein a gradual increase in average program size (over a number of generations) is observed without a corresponding increase in average program fitness [1]. Although there is no consensus on why bloat occurs in genetic programming, under one account, illustrated in Figure 2.1, bloat effectively serves as a defensive mechanism against detrimental mutation or crossover in high-fitness individuals [23].

Bloat can impose serious deleterious effects on genetic programming search, in a variety of manners. Bloat generally increases the computational expense of evaluating program fitness by unnecessarily inflating program execution time, runaway bloat produces programs which impede human understanding, and bloated programs may be grossly overfit to the fitness functions to which they have evolved [1].

A number of bloat reduction mechanisms have been proposed, which range from the simple to the complex. One such method is the imposition of a maximum size on programs, either with regards to depth, breadth, or both. Typically, this constraint is enforced during offspring generation [1, 18]. Unfortunately, this method requires either a good estimate for maximum reasonable program size to be made, or for an adaptive approach to be used during genetic programming run-time [1]. Alternative methods, including those introduced in Poli [24] and Poli and McPhee [25], instead fight bloat by modifying the fitness measure, either effectively discarding bloated programs [24] or imposing a penalty reminiscent of multi-objective-optimization approaches [25]. Yet other proposed techniques seek to prevent the occurrence of bloat during the mutation and crossover stages [26, 27]. A recent and fairly extensive overview of anti-bloat methods is offered in Poli et al. [1].

2.2.4 Application to automated program repair

Weimer et al. [9], Forrest et al. [10], and Le Goues et al. [11] collectively describe "GenProg", a genetic programming system for the automated repair of faulty C programs. GenProg takes as input a codebase containing one or more defects, and a test suite with one or more failing tests reflecting these defects; the number of passing tests in the suite is used as an objective function. The genetic operators employed in GenProg are unusually constrained in that alterations cannot be made to AST subtrees which are run only during passing tests, and in that mutation does not allow for the generation of novel expressions, but instead is limited to the deletion, duplication, or swapping of existing statements within the codebase. Given that crossover also draws on existent subtrees as sources, this means that GenProg is constrained to making use only of the constructions found in the original program. Nevertheless, GenProg is, over a relatively small number of iterations, shown to fully repair defective programs in the majority of cases surveyed [11]. The relationship of the techniques used in this work to the methods employed in GenProg is covered in detail in section 3.3.

Chapter 3

Methodology

In this chapter, a comprehensive account of the methods and techniques used for the work undertaken in this dissertation is given. Care is taken to describe when and how these methods differ from previous work, along with justifications for these differences where applicable. Furthermore, this chapter also describes the means by which the methods have been evaluated.

3.1 Python corpus compilation

In order to easily amass a large and representative collection of Python code, the popular code repository Github [28] was chosen as a source. Github provides free hosting for a number of prominent Python projects, including 13 of the 25 most downloaded packages from PyPI, the centralized index for Python packages [29].

The Github API was used to identify appropriate Python repositories, using the built-in language search parameter. It should be noted that the use of this parameter does not guarantee that the source code in each resultant repository is exclusively or even primarily comprised of the selected language. Because of the importance of including good-quality code in the Python language model, repositories with fewer than 50 "stars" or "forks", both measures of repository popularity on Github [6], were excluded from the results. The resultant search yielded 9429 repositories. Of these, 19 repositories exceeded a 1-hour timeout during attempted download, and were therefore excluded from the corpus, leaving a sum total of 9410 repositories included.

From each of the target repositories, a recursive search through the repository was performed in order to locate files with a "`.py`" ending, so as to attempt to avoid including source code belonging to other languages and non-source files. Allamanis and

Sutton [6] noted that, due to the nature of the preferred contribution flow on Github, many repositories are "forked" (i.e. cloned) from others, it is vital that duplicated code be identified and excluded from corpora, so as not to have a disproportionate influence on language modeling. Whereas Allamanis and Sutton [6] identified forked repositories by comparing SHA hashes of commits to each project, and manually selected what they believed to be the original repository in cases where multiple repositories shared a commit history, deduplication of Python code from this project's corpus was instead achieved by calculating hashes of the contents of each Python module found in each repository, and discarding any duplicate modules.

Finally, in order to allow for parameter optimization and testing of the language model, the Python corpus was randomly split into three training, validation, and test sets on a per-module basis, which constituted 90%, 5%, and 5% of the original corpus, respectively.

3.2 Language modeling

3.2.1 Probabilistic context-free grammar development

The induction of a statistical language model from a large corpus is a highly parallelizable task which lends itself well to implementation via the MapReduce framework [30]; as such, in order to induce a probabilistic context-free grammar from the Python training corpus, a MapReduce job was executed on a 40-node Hadoop cluster[1]. The job used 10000 map tasks and 25 reduce tasks, with a maximum mapper failure percentage of 1% allowed.

Each mapper read in single-line, base-64-encoded Python modules from the corpus, which were then decoded and parsed into ASTs. For every node in the AST, the node type, a JSON-formatted representation of child nodes or values, and a count of 1 were emitted. Additionally, for "`FunctionDef`" and "`ClassDef`" nodes, which have, as child properties, the name of the function or class being defined, an artificial intervening "`Unk`" node was hallucinated and emitted — except in the case of so-called "magic methods" (see Appendix B) — such that in the resultant language model, `FunctionDef` and `ClassDef` production rules were not qualified by name (see Figure 3.1). An example of mapper emissions for a small program can be found in Appendix C.

[1]The use of which was provided by the University of Edinburgh.

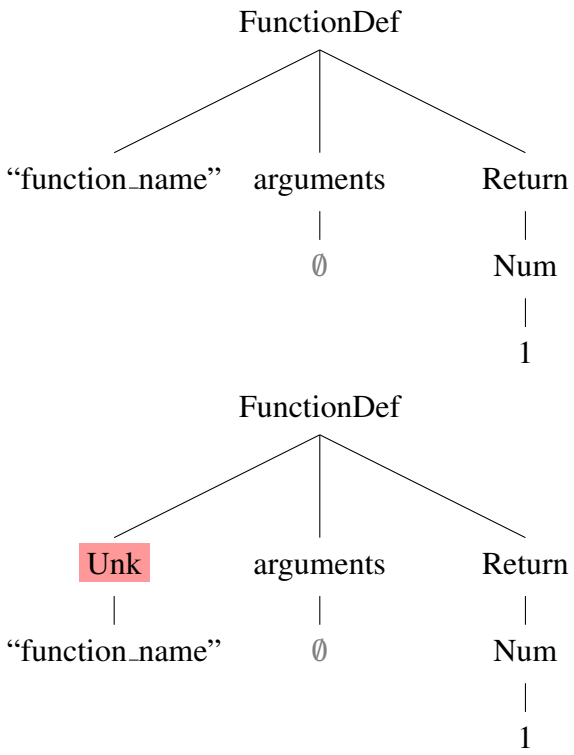


Figure 3.1: Comparison of an unmodified Python AST (top) and the modified version used during PCFG generation, in which an “`Unk`” node is hallucinated (bottom). Note that, in the modified tree, `"function_name"` is no longer a child of the `FunctionDef` node.

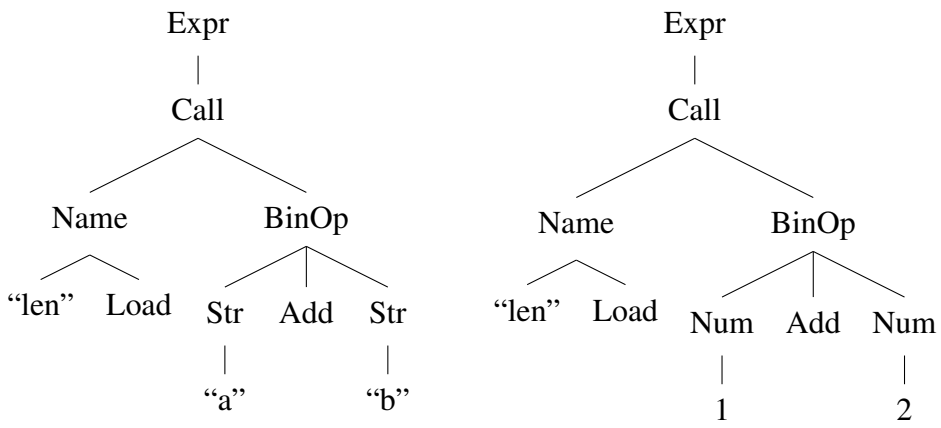


Figure 3.2: Comparison of two Python ASTs corresponding to the expressions `len("a"+"b")` (left) and `len(1+2)` (right). Whereas `len("a"+"b")` will produce a result of `2` when evaluated, `len` will raise a `TypeError` exception when applied to an integer (as in the right case). Note that, in both ASTs, the children of the `Call` node are `Name` and `BinOp`. Further, it should be noted that while, in this example, `BinOp` could be annotated with the types of its operands, were either (or both) operands `Name` nodes, the return type of `BinOp` would be, in the general case, undecidable without program run-time information.

Reducers read in the mapper emissions, as keyed on left-hand-side AST node types, with sum total frequencies for each production rule being calculated and emitted. Subsequently, production rules were divided by left-hand-side nonterminal and sorted by frequency, descending, for later use in AST generation via roulette wheel selection (see subsubsection 3.3.4.2).

3.2.2 Latent variable annotation

The example given in Figure 3.2 serves as a concrete demonstration of a situation in which simple context-free production rules fail to fully capture features of the Python language. As the use of latent nonterminal annotations in statistical language modeling has been shown to lower model perplexity to a degree comparable to manual feature engineering [31], this method was used to develop an annotated version of the PCFG Python language model.

For each nonterminal node, a number of labels was determined by taking a root of the number of production rules with that nonterminal as a left-hand-side, in order to reflect the overall variability of rules for that nonterminal. The 7th root was chosen

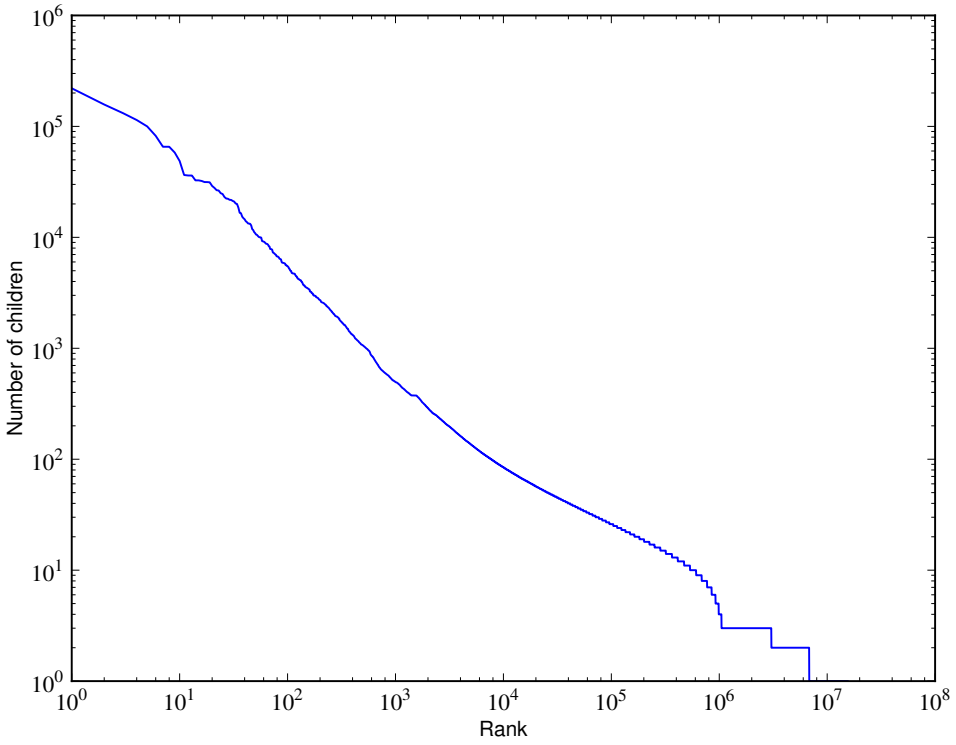


Figure 3.3: PCFG production rules, ranked by number of children. Note that roughly 6.1×10^5 production rules, comprising more than 3.9% of all production rules, have 10 or more children.

as the smallest root for which no nonterminal was assigned a number of labels greater than 10, in order to improve the tractability of model induction. The number of labels assigned for each AST node are documented in Appendix D.

Expectation maximization was used to minimize annotated PCFG perplexity on the training set. As expectation maximization is sensitive to initial conditions and does not guarantee convergence to the global minimum [31], three expectation maximization tasks were run. Due to memory constraints on the available Hadoop cluster, the expectation maximization processes were run in parallel on consumer PCs.

As noted in Matsuzaki et al. [31], even when dynamic programming is used when calculating the most probable node annotations for a given tree, label assignment complexity for a node grows exponentially with the number of node children. In the Python training corpus, it is not uncommon for nodes to have tens, hundreds, thousands, or, in extreme cases, hundreds of thousands of children, often consisting of nodes with the highest search space of possible labels (e.g. `Str` and `Name` nodes). This is visualized

in Figure 3.3. In order to combat this combinatorial explosion, a search limit of 1000 combinations of child label assignments per node was set, with the search initially using the most probable label for each child. Despite this, due to the long[2] run-time of the expectation maximization processes, all three were prematurely halted, before language model convergence, at 20 iterations.

3.2.3 Evaluation of language models

The cross-entropy of a language model M over a token sequence $w_1, ..., w_n$, as measured in bits per token, can be estimated as:

$$H(M) = -\frac{1}{n} \log_2 P(w_1, ..., w_n) \tag{3.1}$$

This measure is commonly used in comparing the relative efficacy of differing language models [13].

For the Python language models developed in this dissertation, cross-entropy was measured on the 5% test data set extracted from the Python corpus as per section 3.1. An additional language model derived from the PCFG model, in which the cumulative probability mass for production rules sharing the same left-hand side was redistributed uniformly across those rules, was used as a baseline measure, with the intention of mimicking the typical random generation methods used in genetic programming.

The cross-entropy was measured for both the PCFG and annotated PCFG models as augmented via simple Laplace smoothing. In addition to this, the cross-entropy of a variant of the PCFG model using add-α smoothing was measured. A value for α was chosen as optimized on the 5% validation data set; because of the high computational expense of measuring a cross-entropy value for each setting of α, Kriging, a surrogate model method from the field of geostatics, was employed in order to reduce the number of measurements taken, with `pykriging` [32] being the specific implementation used [33]. This optimization process is visualized in Figure 3.4.

3.3 Genetic programming system

3.3.1 Overview

As implemented, the genetic programming system takes as inputs a suite of tests compatible with the `pytest` [34] testing framework, and a corresponding faulty Python

[2] Approximately two weeks.

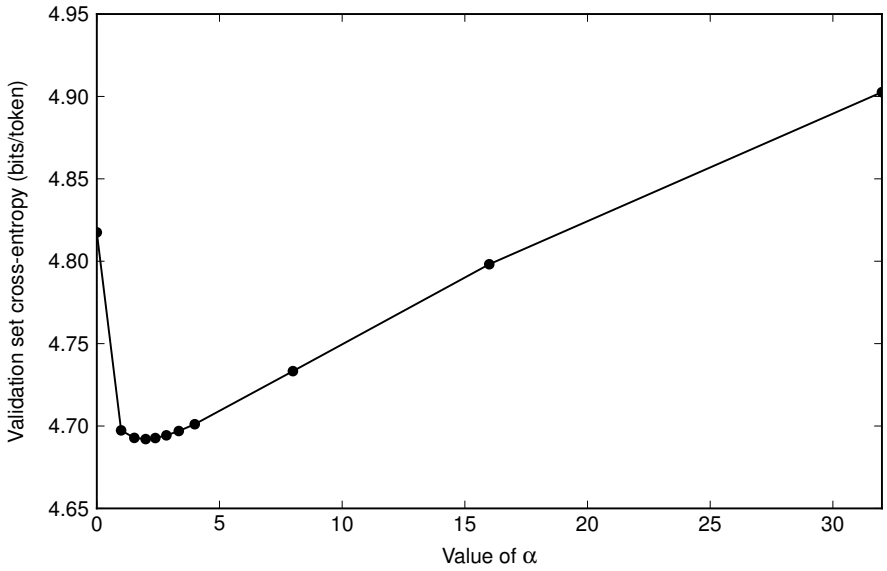


Figure 3.4: Optimization of PCFG additive smoothing parameter via Kriging.

module. For the purposes of this dissertation, repair is limited to a single module, to avoid the need to handle the complications involved in applying genetic operators to individuals comprising multiple distinct ASTs. This restriction still allows for the targeted program and test suite to span multiple modules — but alterations will be restricted to the single module selected. To initialize the search, the test suite is run, and the individual tests divided into positive (passing) and negative (failing) sets. Module execution during the initial test suite evaluation is used to weight each node of the AST (see subsection 3.3.2).

Following on from this, the original module is assigned a fitness. As in Weimer et al. [9], the objective function used as a fitness measurement is a weighted mean of successful tests, with negative tests weighted twice as heavily as positive tests. The domain of the fitness function is $[0, 1]$. If an AST node cannot be successfully serialized in order to be evaluated against the test suite, or in the event of an unhandled exception occurring during the test suite evaluation, a fitness value of 0 is returned. If at any time an AST passes all tests within the test suite, program search is immediately halted, with the AST having been serialized and written to disk.

As is standard in many evolutionary algorithms [2], a hall of fame is initialized, using the input module and its fitness, for later use during crossover (see subsubsection 3.3.4.1). The language model is loaded using the specified probabilistic grammar, and a population of 50 individuals is created by applying mutations to the input module

(see subsubsection 3.3.4.2).

It is at this point that the main loop of the genetic programming system is executed. During each iteration, the fitness of each individual in the population is evaluated, with the best individual being added to the hall of fame. As the code generated may be harmful or arbitrarily computationally expensive, the test suite is run in a sandbox provided by a modified version of `pysandbox` [35], a Python library which restricts program access to the host system, maximum program recursion depth, and maximum program run-time. After test suite evaluation is completed, the language model is updated with respect to the population and corresponding fitnesses (see subsection 3.3.3). An average AST size for the population is also computed, for later use in bloat prevention (see subsection 3.3.5). A new population is then initialized, with the top 50% of individuals from the previous generation being retained, as in Forrest et al. [10]. The remaining individuals are generated via crossover and mutation. This process is repeated for a maximum of 250 iterations, or until an individual with fitness 1 has been generated. If no such individual has been found, an interactive prompt allows for the inspection and writing to disk of members of the last population and hall of fame.

3.3.2 Fault localization

In order to constrain the repair search space and better direct the GenProg search process, Weimer et al. [9] employ a rudimentary fault localization scheme. In the method utilized for GenProg, before search begins, AST subtrees are weighted based on their execution while running the test suite. When a subtree is executed only during passing tests, it is excluded from modification during the search process. If a subtree is executed during both failing and passing tests, it receives a weight of 0.1. If a subtree is executed exclusively during failing tests, it receives a weight of 1.0. Subsequently, during the search process, mutation and crossover select target nodes proportionally to their weightings, i.e. a subtree with purely negative test execution is ten times more likely to be mutated than one with mixed execution. It is important to note that, under this method, subtrees with a weight of 0 may still be used as the source of a crossover or mutation operation.

The genetic programming system presented in this dissertation differs from the above-described method in two significant ways. Firstly, because of the difficulty of determining runtime execution on a per-AST-node level in Python, the lines run during successful and unsuccessful test cases are instead recorded. When an AST is con-

structed from the program source, most nodes are tagged with their corresponding line number, where applicable. Node weightings are then applied on a per-line basis. In cases where a node does not have a line number attached to it, the system attempts to assign the weight of the closest ancestor node. Should no such weight exist, the node is given a weight of 0.

Secondly, in evaluating the line weightings assigned using the GenProg localization method, it soon became clear that the method was too coarse-grained to reliably direct genetic programming search, when applied to the test cases selected: for many programs, no lines were executed purely during failing tests, and the vast majority of lines were executed during both positive and negative tests. In order to combat the resultant relatively uniform weighting of nodes and encourage the alteration of likely faults, line weights were instead assigned such that:

$$w_s = \frac{1}{|T_P| + |T_N|} \sum_{t \in T_N} \begin{cases} 1 & \text{if } s \in t \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

where w_s is the weight given to the AST nodes corresponding to line s, and T_P and T_N are the positive and negative test case sets, respectively. A comparison between the weightings assigned under this scheme and those assigned using the GenProg method is presented in Figure 3.5. As can be seen, whereas the GenProg fault localization method distributes equal alteration selection probability mass across all but one line of the sample program, the approach used in this dissertation makes more varied weight assignments and, in this case, assigns the greatest selection probability to the line in which a fault was induced. It should be noted that, under both fault localization and weighting methods, the number of times a line or node is executed during any one given test beyond its initial execution does not have any bearing on its subsequent weight.

3.3.3 Language model adaptation

In order to adapt the language model used by the genetic programming system to the task at hand during run-time, ant colony optimization was performed over the production rules. This also allowed for the inclusion of production rules from the task's original source code which might otherwise not be present in the language model, e.g. variable names.

The initial weights of production rules were drawn from their frequencies within the chosen language model. After the evaluation of each generation, these production

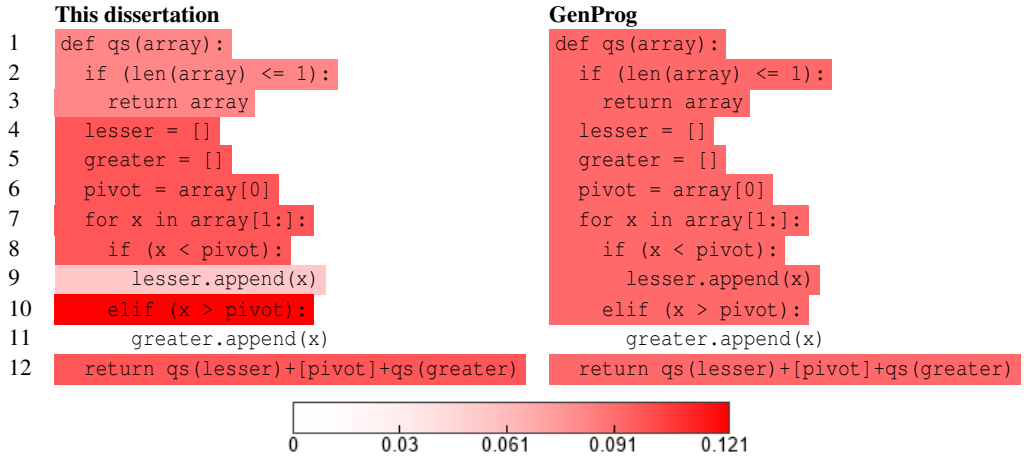


Figure 3.5: Comparison of line alteration selection probabilities, using the fault localization approach applied in this dissertation and the approach utilized by GenProg. The program shown is a QuickSort implementation in which a fault has been induced in line 10, with a corresponding test suite including 9 passing tests and 3 failing tests.

rule weights were updated as follows:

$$\tau_{ij}^{t+1} = (1-\rho)\tau_{ij}^{t} + \sum_{k \in K_t} \begin{cases} \frac{1}{1-f(k)} & \text{if } i \to j \in k \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

where τ_{ij}^{t} is the weighting τ of the production rule $i \to j$ at iteration t, ρ is a constant evaporation rate within $[0,1]$, K_t is the population at iteration t, and f is the objective function.

This equation can be thought of as comprising two phases: the "evaporation" step, and the "deposition" step. During evaporation, the previous weighting for a production rule is diminished in accordance with ρ, decreasing its relative likelihood of being used for a subsequent mutation. By adjusting the value of ρ, the intensification of the ACO search process can be tuned; $\rho = 1$ will result in only the production rules found in the latest generation being available during the construction of the next generation, whereas under $\rho = 0$ initial language model weightings will never be diminished. Empirical work has suggested a good general range for ρ of $[0.01, 0.2]$ [36]; a value of 0.05 was chosen for the genetic programming system presented in this dissertation. The effect of evaporation on the initial weightings of PCFG `Name` node production rules is shown in Figure 3.6.

During the deposition step, each production rule weight is increased based on the fitnesses of the individuals which have used that production rule. Specifically, a distance is calculated from the objective function f, which has a domain $[0,1)$; the amount

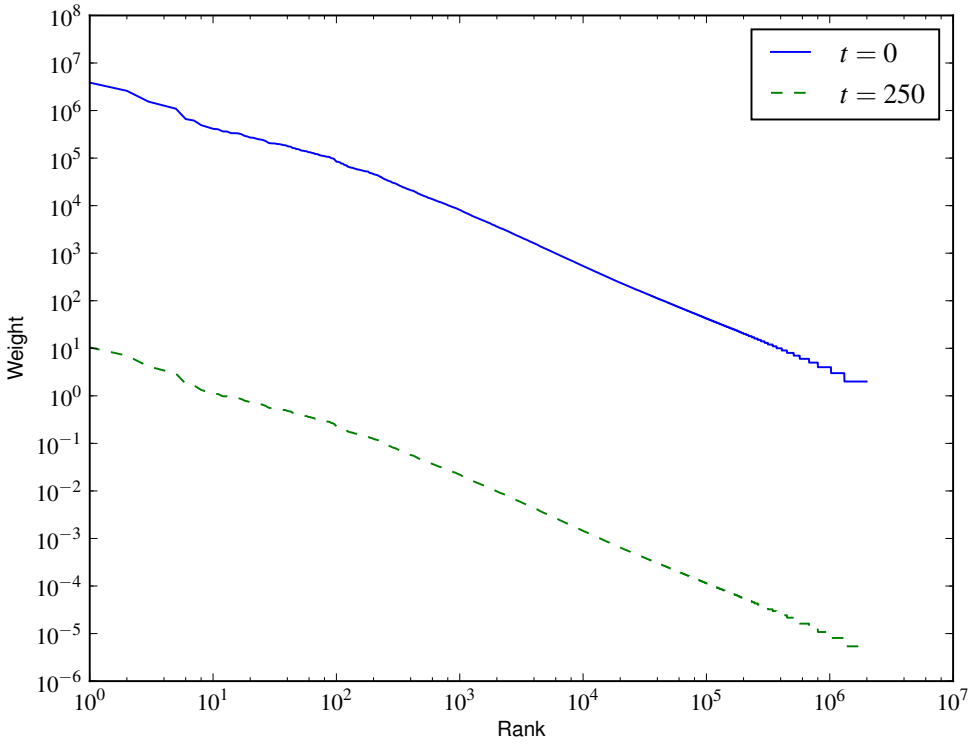


Figure 3.6: `Name` node production rule weightings at iterations t with $\rho = 0.05$, assuming no deposition. Production rules are ranked by initial frequency.

of weight deposited by each individual to its constituent production rules is then inversely proportional to this distance. Figure 3.7 shows the cumulative effects of the evaporation and deposition processes on the selection probability of a novel `Name` production rule.

It should be noted that this approach to language model adaptation deviates from typical ACO in that, whereas typical ACO path weightings (as weighted by a parameter α) are interpolated with a heuristic factor (as weighted by a parameter β) during the construction of a new path, the choice of a production rule during mutation within the genetic programming system is performed using the path weightings exclusively. This is equivalent to a choice of heuristic weighting parameter $\beta = -\infty$. The decision to eschew the inclusion of a heuristic factor was made in order to avoid the need for optimization of the weights α and β, the values of which can have a large effect on the ACO search process [37].

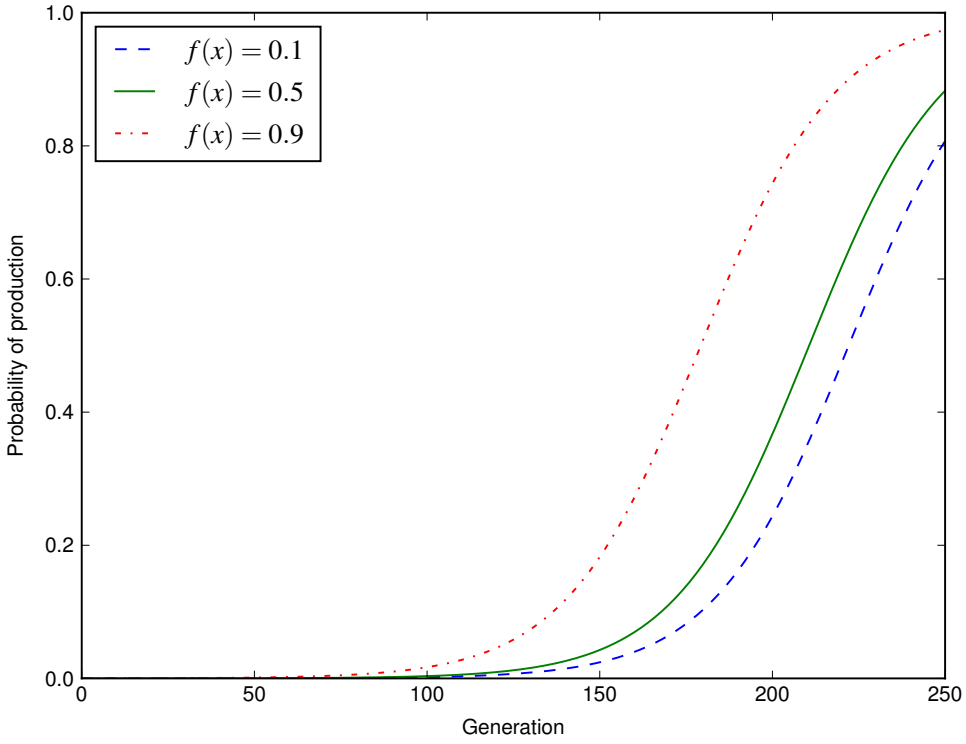


Figure 3.7: Production probability of a previously unseen `Name` node, without the reinforcement of competing `Name` production rules, under varying average population fitnesses. $\rho = 0.05$; $|K| = 50$. As can be seen, after 250 generations, the influence of the initial language model on `Name` production rule selection is greatly diminished.

3.3.4 Genetic operators

In order to create new individuals (and, hence, explore the search space) during genetic programming, two biologically-inspired genetic operators, crossover and mutation, are employed.

3.3.4.1 Crossover

Input :
an array of individuals K;
an objective function $f : k \rightarrow \mathbb{R}$;
a numerical parameter $p \in [0,1]$
Output: an individual $k \in K$
Let $n = max(\lfloor p \cdot |K| \rfloor, 1)$;
begin
  Shuffle(K);
  for $i = \{1, ..., n\}$ **do**
    $c_i \longleftarrow K_i$;
  end
  $k \longleftarrow \arg\max_{k \in c} f(k)$;
  return k;
end

Algorithm 1: Tournament selection

In order to generate individuals for a new generation, crossover is performed using individuals from the previous generation in conjunction with members of the hall of fame (see subsection 3.3.1). Crossover begins with the choice of two members of this pool, using tournament selection (see Algorithm 1). It should be noted that no check is performed to ensure the selection of two different members of the population.

During crossover, a subtree of a sender tree replaces a subtree of a receiver tree. Both individuals take turns as sender and receiver. Whereas the subtree selected from the sender is chosen arbitrarily from the set of all sender AST nodes, the receiver node targeted for replacement is selected proportionally to node weightings (see subsection 3.3.2) using roulette wheel selection (see Algorithm 2). An example of a crossover replacement is given in Figure 3.8. After replacement, the parent node of the replacement location is checked, in order to ensure that it remains within the grammar defined

Input : an array of $(weight, item)$ tuples X;
Output: a choice $x \in X$
Let $i = 0$;
Let $sum = 0$;
Choose $target \sim U(0, \sum_{w,x \in X} w)$;
begin
  while $sum < target$ **do**
    $i = i + 1$;
    $(w, x) \longleftarrow X_i$;
    $sum = sum + w$;
  end
  return x;
end

Algorithm 2: Roulette wheel selection

by the language model [3]. If the corresponding production rule is found within the language model, the replacement is accepted regardless of probability. Finally, all nodes within the replacement subtree are re-weighted to match the weight of the parent node.

As the process described above has a failure condition (i.e. cases in which there are no valid substitutions which can be made with regards to the language model), for both assignments of sender and receiver, crossover is attempted a limited number of times. In this way, the overall crossover operation returns between two and zero offspring. This is compensated for in the genetic programming system by the repeated selection of crossover candidates until a desired population size has been reached.

3.3.4.2 Mutation

Mutation is applied to each individual produced through crossover with a probability $P(m) = 0.1$, the same rate used for the mutation of statements executed during both positive and negative test cases in Weimer et al. [9].

When an individual is mutated, first, a target node is selected proportionally to node weightings (see subsection 3.3.2) using roulette wheel selection (see Algorithm 2). In using this method of selection, parts of the program identified by the fault localization scheme to be more likely to contain a defect are likewise more likely to undergo alteration. Following this, the language model is used to generate a new node of the same type. In doing so, a production rule is selected from the language model using

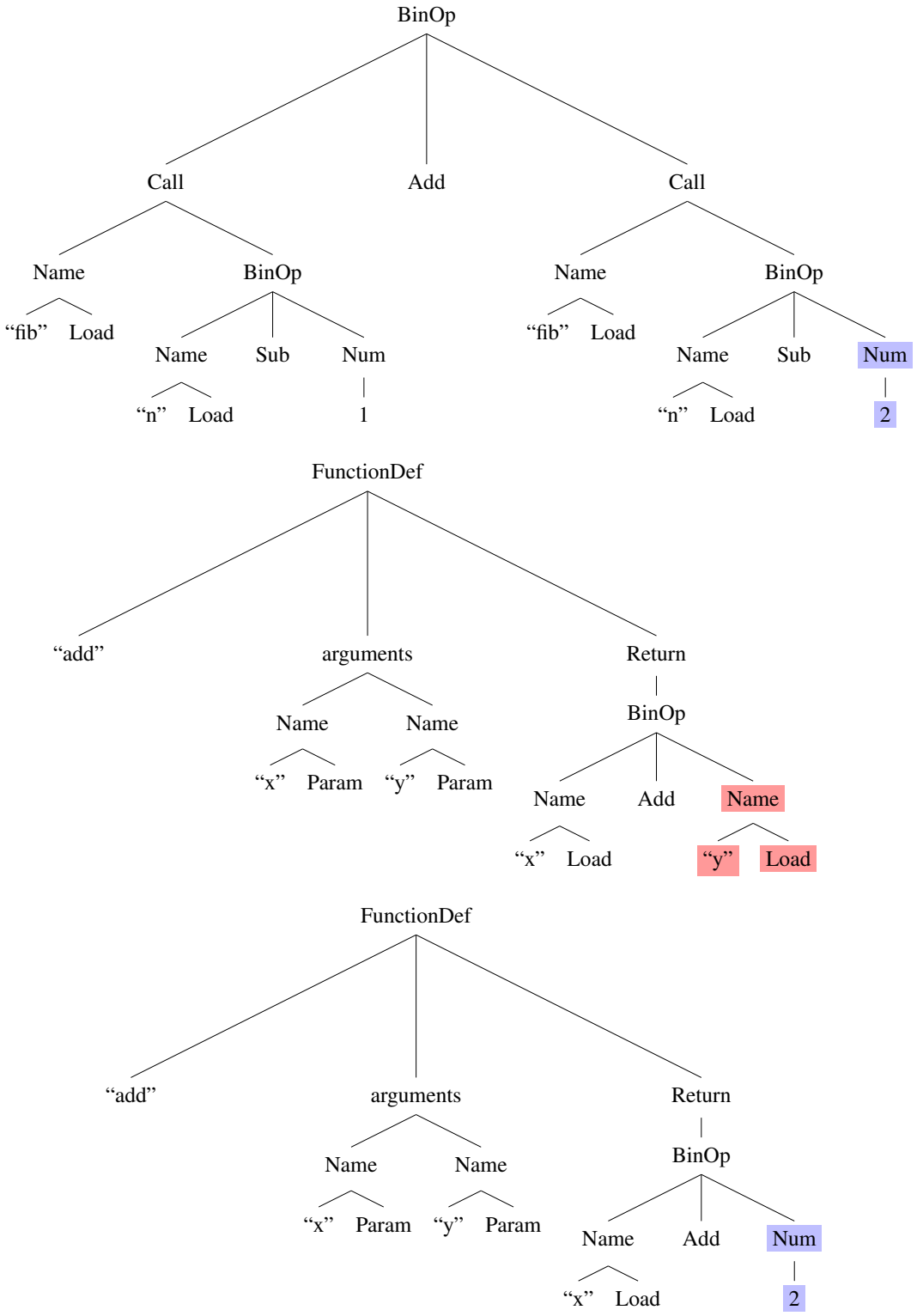


Figure 3.8: An example of the crossover operator as applied to two ASTs. A subtree of the sending parent (top), `Num(2)`, replaces a subtree of the receiving parent (middle), `Name("y", Load)`, to produce an offspring (bottom).

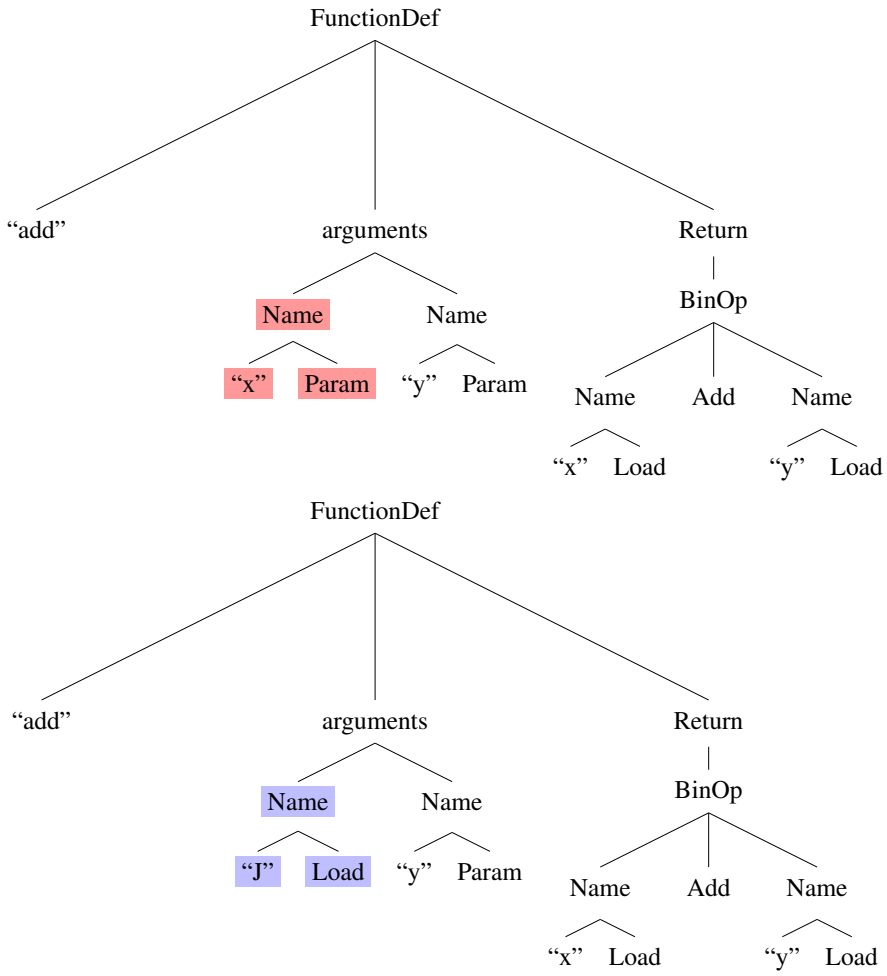


Figure 3.9: An example of the mutation operator. A subtree from the original individual (top), `Name("x", Param)`, is replaced with a new node of the same type (as drawn from the language model), `Name("J", Load)`, to produce the mutant (bottom).

roulette wheel selection. If the production rule contains nonterminals (i.e. further AST nodes), these are generated recursively using the language model. Because this process potentially allows for the generation of AST subtrees of unbounded size, in order to limit the computational expense of mutation and to restrict program bloat, a maximum recursion depth limit of 8 is imposed. This limit is sufficiently deep to allow for the successful generation of all node types. Should the generation of a new subtree from the language model exceed this recursion depth constraint, an exception is raised and no mutation is applied to the individual. Otherwise, after a new node of the requisite type has been generated, the target node is replaced with the new node. Subsequently, a check is performed on the new node's parent, to ensure that the operation has resulted in a tree which conforms to the grammar defined by the language model. Finally, the

new node and all descendants are re-weighted using the parent node's weight value. An example of mutation is given in Figure 3.9.

3.3.5 Bloat compensation

As mentioned in subsection 2.2.3, the use of parsimony pressure in combating bloat is essential to ensuring the success of genetic programming. While the use of sandboxing during program test evaluation limited the maximum recursion depth and overall maximum run-time for each individual, bloat has been noted as leading to overfitting of a program to the objective function and as increasing the expense of crossover, mutation, and grammar adaptation[3] [1]. Further to the excess computational cost caused by bloat during language model adaptation, bloated individuals have the potential to exert a disproportionate effect on production rule weightings during language model adaptation — especially in cases where bloated subtrees have been replicated throughout the population.

For the genetic programming system presented in this dissertation, a variant of the method introduced in Poli [24] was employed. Before the population of each new generation, the average size of the individuals in the previous generation, as given by number of nodes per tree, is measured. Subsequently, before a newly generated individual is to be added to the new generation, its size is measured. If that size is larger than the previous population average, with a random probability $P(r)$, the individual is discarded and does not enter the new population. In this dissertation, a value of $P(r) = 0.025$ was chosen through a process of iterative program size observation and manual value optimization.

This method differs from that presented in Poli [24] in that the elimination process occurs before the addition of an individual to a new population rather than before individual fitness evaluation. This choice was made in order to maintain a constant population size of individuals that passed the bloat check (this is to say, in order to avoid a potential population bottleneck).

3.3.6 Evaluation of genetic programming system

In order to evaluate the efficacy of the genetic programming system, first, a number of codebases had to be assembled in which defects could be induced. These codebases had to fulfill the following criteria:

[3] In the case of this dissertation, language model adaptation.

1. Well-tested Python code, with all tests passing and compatible with the `pytest` testing framework
2. No external dependencies or requirements
3. Written largely functionally, with few or no IO operations, network interactions, or other side-effects
4. Not included within the Python corpus used for language modeling.

Using these criteria, five code repositories were manually selected; further to these, five supplementary hand-written repositories were used as test cases. In each of these codebases, one fault was induced such that one more more test cases no longer were passing; in the case of one of the selected codebases (the "chess" task, see Appendix E, subsection E.2.3), this fault represented the re-introduction of a previously repaired bug, as taken from the repository's commit history. The genetic programming system, using the Python language model, was evaluated in its performance in repairing these codebases against a baseline genetic programming system, which used the "uniform" language model described in subsection 3.2.3. It should be noted that this choice in initial production weightings between the two systems evaluated constituted the only difference between them; both used the ACO-based language model update method described in subsection 3.3.3.

Chapter 4

Results, discussion, and further work

In this chapter, the results of the corpus compilation, language modeling, and genetic programming system tasks are evaluated and discussed in relation to previous work. Suggestions for improvements are also made, and potential areas for future research building upon this work highlighted.

4.1 Python corpus

A total of 9407 repositories were included in the Python corpus. From these, 620477 modules were extracted, totaling 4.9239 gigabytes in size. As measured using Python's built-in tokenization module, the training, validation, and test sets contained a total of 850.1 megatokens. While these results fall shy of the size of the Java corpus presented in Allamanis and Sutton [6], the Python corpus constitutes a significant advance on the size of Python corpora used in previous work [38, 39, 40, 41, 42]. An in-depth analysis of the corpus — aside from the language modeling performed — fell outside the scope of this project, but provides an avenue for future work.

4.2 Language modeling

The performances of the baseline language model, PCFG with Laplace smoothing, PCFG with add-α smoothing, and best annotated PCFG model (as measured on the validation data set) are compared in Table 4.1. As can be seen, while all three models learned outperform the baseline, the PCFG with Laplace smoothing displays the best performance, with an estimated cross-entropy of 4.84 bits per token. This is a result competitive with the 4.9 bits per token result reported in Allamanis and Sutton [6],

Table 4.1: Language model comparison on test set

Model	**Cross-entropy (bits/token)**
Uniform PCFG	8.61
PCFG (Laplace smoothing)	4.84
PCFG (Additive smoothing)	5.11
Annotated PCFG (best of 3)	5.78

although a direct comparison cannot be made without an estimate of the relative true per-token entropies of Java and Python — establishing this relationship by learning a trigram model over the Python corpus, comparable to that learned over the Allamanis and Sutton [6] Java corpus, is a potential area for future work.

As can be seen in Figure 3.4, the Kriging process as applied to the validation data set found an optimal value of $\alpha = 2$; however, as evaluated on the test data set, this parameterization resulted in poorer performance than simple Laplace smoothing. As can additionally be seen from this figure, the performance of the Laplace-smoothed PCFG on the validation set was 4.7 bits per token. The disparity between these results and those observed during test set evaluation are potentially suggestive of substantial differences between the test and validation sets. Further work may wish to explore whether these sets should be expanded in size, in order to perhaps make them more congruent (or representative of the training set).

The best of the three annotated PCFG models learned, employing Laplace smoothing, displayed an order of magnitude poorer performance on the test set than the unannotated, Laplace-smoothed PCFG. One possible explanation for this is that the annotated model, as produced, is severely underfit, given that the expectation maximization process used for parameter optimization was prematurely halted. An obvious opportunity for further work would be to re-attempt this optimization process, allowing for convergence of the EM process — however, the amount of time necessary to accomplish this is unclear.

An alternative explanation for the poor performance of the annotated model is the comparatively low number of annotations used: work from the field of natural language processing has found larger model sizes to demonstrate markedly improved parsing performance [31, 43]. Whereas, for example, Matsuzaki et al. [31] identified a choice of 16 annotation labels per original PCFG node to give the best performance of the choices surveyed in that work, yielding a grammar size on the order of tens of millions

of symbols, an average of 2.14 annotation labels per nonterminal were used in this dissertation, yielding a total of 188 nonterminal symbols. Given the issues with the tractability of learning a parameterization of a PCFG model given a grammar of this size already observed during this project (see subsection 3.2.2), increasing model size presents an obvious problem. One solution to this would be to attempt annotation via manual feature selection, or to attempt to learn features via another means entirely.

When reading from the Python corpus during both annotated and unannotated PCFG induction, 429 megabytes worth of modules failed to be parsed into ASTs; additionally, 0.08% of mappers failed unexpectedly during the execution of the PCFG MapReduce job. The two most likely contributors to these failures are syntactically invalid Python files and modules written in Python 3, which has introduced new syntax incompatible with the version of Python (2.7.5) used for this project [44]. While these failures reflect a loss of less than 10% of corpus data, a more sophisticated approach in which an additional language model is trained for Python 3 alongside the Python 2 model could be a worthwhile future pursuit.

4.3 Genetic programming system

Appendix E presents the faults induced in order to create evaluation tasks for the genetic programming system. Both a baseline system, which utilized the "uniform" language model, and a system utilizing the PCFG language model were evaluated against these tasks. The results of this evaluation are shown in Table 4.2. As can be seen, the PCFG-based system was able to improve fitness in four out of ten tasks, with the baseline system improving fitness in one task alone. This result suggests that the use of a PCFG in guiding system search proved beneficial.

While, owing to the fact that all but one of the repair tasks used for evaluation in this work incorporated hand-written defects, a direct comparison between the systems cannot be drawn, one clear weakness of the automated program repair methodology presented in this dissertation is that the rate of successful repairs falls well short of the 52% success rate reported in Le Goues et al. [11]. A number of factors could account for this: firstly, Le Goues et al. [11] focused on the repair of C programs, which could differ substantially in difficulty from the repair of Python programs. Additionally, while the results of the genetic programming system in this dissertation appear to support a PCFG-based approach as being more effective than initially uniform production rule weightings, GenProg does not use a language model whatsoever, but, instead,

		Final fitness (max)	
Task	**Initial fitness**	**Baseline**	**PCFG**
add_one	0.00	0.00	**1.00**
even	0.41	**0.50**	**0.71**
quicksort_gt	0.60	0.60	0.60
quicksort_name	0.20	0.20	**1.00**
quicksort_name_gt	0.09	0.09	0.09
astoptimizer	0.80	0.80	0.80
checkers	0.77	0.77	0.77
chess	0.95	0.95	**0.98**
python-dis3	0.31	0.31	0.31
sudoku	0.67	0.67	0.67

Table 4.2: Comparison of the baseline and PCFG-model-based genetic programming systems, with improvements to initial task fitnesses highlighted. Tasks are split into hand-written (top) and real-world (bottom) categories.

exclusively utilizes existing statements from within the defective system — whether the use of a language model for program repair is appropriate in the first place is a question left unanswered by this dissertation. One potential means of improving a language-model-based approach — and a method that could potentially constitute an advantage over the approach used in GenProg — would be the learning and use of a domain-specific (or even project-specific) language model, to avoid the long "ramp-up" time of likely novel rule production apparent from Figure 3.7.

It should be noted that, of the ten tasks on which it was evaluated, the PCFG model was only able to repair all test cases in two instances; in the other two instances, one or more of the original negative test cases was repaired, but new faults were introduced, causing originally positive test cases to fail. One such instance, in which a fault was introduced in the "chess" task, is represented in Figure 4.1. These instances almost certainly reflect genetic programming system runs in which a beneficial mutation, which improved program fitness by repairing one or more negative test cases, occurred in an individual which bore a harmful mutation, causing the failure of one or more positive test cases — given that the fitness function used weighs negative test case repair more strongly than continued success of positive test cases, such an individual has a higher-than-initial fitness, and will be highly selected for during the construction of

```
    (Preceding lines omitted)
       def can_move(self, source, destination):
         if ((self.color == 'white') and \
             (x.index(source[0]) == x.index(destination[0]))):
-          ordinary_move = (abs((y.index(destination[1]) \
+          ordinary_move = (int((y.index(destination[1]) \
                         - y.index(source[1]))) == 1)
           two_square_move = not self.moved and \
             y.index(destination[1]) - y.index(source[1]) == 2
           return (ordinary_move or two_square_move)
    (Lines omitted)
         if ((not self.is_tutorial_mode()) and \
             self.__is_player_in_check(('white' if \
             (self.turn == 'black') else 'black'))):
           self.__switch_turn()
           self.check = self.turn
-          return
+          return self.__switch_turn()
         self.__switch_turn()
```

Figure 4.1: Example in which the PCFG-model-based genetic programming system improves fitness by fixing a negative test (beginning lines), while also breaking a positive test (ending lines).

subsequent generations. A potential solution to the introduction of new faults, as in these instances, would be a postprocessing step in which each permutation of changes found in the most meritorious individual is evaluated against the test suite, to attempt to determine a minimal change set containing only true repairs. A more sophisticated approach to fault localization, or the introduction of a minimum weight threshold for node alteration, might also reduce the incidence of novel fault introduction.

Chapter 5

Conclusion

This dissertation sought to determine whether a pre-trained language model could feasibly be incorporated into a genetic programming system, and whether the use of a language model to direct genetic programming search would bear any advantages over traditional approaches. In order to do so, a large, multi-megatoken corpus of the Python programming language was assembled, two language models were induced from this corpus, a genetic programming system incorporating one of these language models was implemented, and this genetic programming system was evaluated on the task of automated program repair. The results of the evaluation support the notion that the use of a pre-trained language model is advantageous, when compared to the induction of a language model during the search process alone, but leaves open the question of whether language-model-based genetic programming is well-suited to the task of program repair. The dissertation also highlights a number of opportunities for future research — most notably, whether further experimentation and investment of effort into the language modeling process might yield improved genetic programming results.

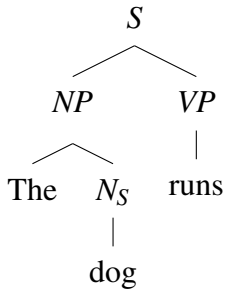
Appendix A

Worked PCFG example

Consider the simple probabilistic grammar with production rules:

$$
\begin{aligned}
S &\rightarrow NP\ VP\ [1.0] \\
NP &\rightarrow N_C\ [0.5] \\
NP &\rightarrow \text{The}\ N_S\ [0.5] \\
N_C &\rightarrow \text{Water}\ [1.0] \\
N_S &\rightarrow \text{person}\ [0.25] \\
N_S &\rightarrow \text{dog}\ [0.75] \\
VP &\rightarrow \text{runs}\ [1.0]
\end{aligned}
$$

Under this, the sequence "The dog runs" would be parsed:



giving the probability:

$$
\begin{aligned}
P(\text{“The dog runs”}) &= P(S \rightarrow NP\ VP) \cdot P(NP \rightarrow \text{The}\ N_S) \cdot P(N_S \rightarrow \text{dog}) \cdot P(VP \rightarrow \text{runs}) \\
&= 1.0 \cdot 0.5 \cdot 0.75 \cdot 1.0 \\
&= 0.375
\end{aligned}
$$

Appendix B

List of Python magic methods

As listed in Kettler [45], the "magic" method names which carry special significance under Python's object model are as follows:

- `__abs__`
- `__add__`
- `__and__`
- `__bool__`
- `__bytes__`
- `__call__`
- `__ceil__`
- `__cmp__`
- `__coerce__`
- `__complex__`
- `__contains__`
- `__copy__`
- `__deepcopy__`
- `__del__`
- `__delattr__`
- `__delete__`
- `__delitem__`
- `__dir__`
- `__div__`
- `__divmod__`
- `__enter__`
- `__eq__`
- `__exit__`
- `__float__`
- `__floor__`
- `__floordiv__`
- `__format__`
- `__ge__`
- `__get__`
- `__getattr__`
- `__getattribute__`
- `__getinitargs__`
- `__getitem__`
- `__getnewargs__`
- `__gt__`
- `__hash__`
- `__hex__`
- `__iadd__`
- `__iand__`
- `__idiv__`
- `__ifloordiv__`
- `__ilshift__`
- `__imod__`
- `__imul__`
- `__index__`

- `__init__`
- `__instancecheck__`
- `__int__`
- `__invert__`
- `__ior__`
- `__ipow__`
- `__irshift__`
- `__isub__`
- `__iter__`
- `__itruediv__`
- `__ixor__`
- `__le__`
- `__len__`
- `__long__`
- `__lshift__`
- `__lt__`
- `__missing__`
- `__mod__`
- `__mul__`
- `__ne__`
- `__neg__`
- `__new__`
- `__nonzero__`
- `__oct__`
- `__or__`
- `__pos__`
- `__pow__`
- `__radd__`
- `__rand__`
- `__rdiv__`
- `__rdivmod__`
- `__repr__`
- `__reversed__`
- `__rfloordiv__`
- `__rlshift__`
- `__rmod__`
- `__rmul__`
- `__ror__`
- `__round__`
- `__rpow__`
- `__rrshift__`
- `__rshift__`
- `__rsub__`
- `__rtruediv__`
- `__rxor__`
- `__set__`
- `__setattr__`
- `__setitem__`
- `__sizeof__`
- `__str__`
- `__sub__`
- `__subclasscheck__`
- `__truediv__`
- `__trunc__`
- `__unicode__`
- `__xor__`

Of these, `__bool__` and `__bytes__` are specific to Python 3.

Appendix C

Example of PCFG mapper emissions

Shown below is a Python implementation of a function for computing the Fibonacci numbers, followed by the corresponding output of the PCFG mapper task for same. Where an emission would exceed the page width, the line has been wrapped and indented.

```
def fib(n):
  if n <= 1:
    return 1
  return fib(n - 1) + fib(n - 2)
```

```
!Module {"body":["!FunctionDef"]} 1
!Unk fib 1
!FunctionDef {"args":"!arguments","body":["!If","!Return"],
    "decorator_list":[],"name":"!Unk"} 1
!arguments {"args":["!Name"],"defaults":[],"kwarg":null,
    "vararg":null} 1
!If {"body":["!Return"],"orelse":[],"test":"!Compare"} 1
!Return {"value":"!BinOp"} 1
!Name {"ctx":"!Param","id":"n"} 1
!Compare {"comparators":["!Num"],"left":"!Name","ops":["!LtE"]} 1
!Return {"value":"!Num"} 1
!BinOp {"left":"!Call","op":"!Add","right":"!Call"} 1
!Param {} 1
```

```
!Name {"ctx":"!Load","id":"n"} 1
!LtE {} 1
!Num {"n":1} 1
!Num {"n":1} 1
!Call {"args":["!BinOp"],"func":"!Name","keywords":[],
    "kwargs":null,"starargs":null} 1
!Add {} 1
!Call {"args":["!BinOp"],"func":"!Name","keywords":[],
    "kwargs":null,"starargs":null} 1
!Load {} 1
!Name {"ctx":"!Load","id":"fib"} 1
!BinOp {"left":"!Name","op":"!Sub","right":"!Num"} 1
!Name {"ctx":"!Load","id":"fib"} 1
!BinOp {"left":"!Name","op":"!Sub","right":"!Num"} 1
!Load {} 1
!Name {"ctx":"!Load","id":"n"} 1
!Sub {} 1
!Num {"n":1} 1
!Load {} 1
!Name {"ctx":"!Load","id":"n"} 1
!Sub {} 1
!Num {"n":2} 1
!Load {} 1
!Load {} 1
```

Appendix D

Annotated PCFG nonterminal labels

Nonterminal	Labels	Nonterminal	Labels	Nonterminal	Labels
Add	1	GeneratorExp	1	Pass	1
And	1	Global	3	Pow	1
Assert	1	Gt	1	Print	2
Assign	2	GtE	1	RShift	1
Attribute	7	If	5	Raise	1
AugAssign	2	IfExp	2	Repr	1
BinOp	2	Import	1	Return	1
BitAnd	1	ImportFrom	5	Set	2
BitOr	1	In	1	SetComp	1
BitXor	1	Index	1	Slice	2
BoolOp	3	Invert	1	Store	1
Break	1	Is	1	Str	9
Call	4	IsNot	1	Sub	1
ClassDef	4	LShift	1	Subscript	1
Compare	2	Lambda	1	TryExcept	4
Continue	1	List	3	TryFinally	3
Del	1	ListComp	1	Tuple	3
Delete	1	Load	1	UAdd	1
Dict	4	Lt	1	USub	1
DictComp	2	LtE	1	UnaryOp	1
Div	1	Mod	1	Unk	7
Ellipsis	1	Module	5	While	3
Eq	1	Mult	1	With	3
ExceptHandler	3	Name	8	Yield	1
Exec	1	Not	1	alias	4
Expr	1	NotEq	1	arguments	4
ExtSlice	1	NotIn	1	comprehension	2
FloorDiv	1	Num	6	keyword	5
For	4	Or	1		
FunctionDef	6	Param	1		

Appendix E

Faults induced in evaluation tasks

E.1 Hand-written cases

These cases have been implemented specifically for use in evaluating the genetic programming system. For each, the module implemented is shown in full.

E.1.1 add_one

```
    add_one(x):
-     return x + 1
+     return x + 0
```

E.1.2 even

```
    def is_even(n):
      n = abs(n)
      if n == 0:
        raise ValueError("N cannot be 0")
      if n == 1: return False
-     if n == 2: return True
+     if n == 2: return False
      if n == 3: return False
-     if n == 4: return True
-     if n == 5: return False
-     if n == 6: return True
+     if n == 4: return False
+     if n == 5: return True
+     if n == 6: return False
      if n == 7: return False
      if n == 8: return True
      return is_even(n - 8)
```

E.1.3 quicksort_gt

```
    def sort(array):
      if len(array) <= 1:
        return array
      less = []
      greater = []
      pivot = array[0]
      for x in array[1:]:
        if x < pivot:
          less.append(x)
-       elif x >= pivot:
+       elif x > pivot:
          greater.append(x)
      return sort(less) + [x] + sort(greater)
```

E.1.4 quicksort_name

```
    def sort(array):
      if len(array) <= 1:
        return array
      less = []
      greater = []
      pivot = array[0]
      for x in array[1:]:
        if x < pivot:
          less.append(x)
        elif x >= pivot:
          greater.append(x)
-     return sort(less) + [x] + sort(greater)
+     return sort(less) + [pivot] + sort(greater)
```

E.1.5 quicksort_name_gt

```
    def sort(array):
      if len(array) <= 1:
        return array
      less = []
      greater = []
      pivot = array[0]
      for x in array[1:]:
        if x < pivot:
          less.append(x)
-     elif x >= pivot:
+     elif x > pivot:
        greater.append(x)
-   return sort(less) + [x] + sort(greater)
+   return sort(less) + [pivot] + sort(greater)
```

E.2 Real-world cases

These cases are taken from publicly-available online repositories with pre-existing tests, as adapted to integrate with the genetic programming system. For each, in addition to the line or lines altered in order to induce a fault, the URL of the repository and a brief summary of the project are shown.

E.2.1 astoptimizer

```
    (Preceding lines omitted)
    if PYTHON3:
      INT_TYPES = (int,)
    else:
-     INT_TYPES = (int, long)
+     INT_TYPES = (int,)
    FLOAT_TYPES = INT_TYPES + (float,)
    COMPLEX_TYPES = FLOAT_TYPES + (complex,)
    IMMUTABLE_TYPES = COMPLEX_TYPES + STR_TYPES + (bool, NONE_TYPE)
    (Following lines omitted)
```

Repository URL: `https://bitbucket.org/haypo/astoptimizer`

A project offering safe compile-time optimization of Python ASTs.

E.2.2 checkers

(Preceding lines omitted)

```
        if piece in active_pieces:
          if self.board.valid_jumps(idx, opponents):
            moves.append((idx, True))
-         elif self.board.valid_moves(idx):
-           moves.append((idx, False))
+       elif self.board.valid_moves(idx):
+         moves.append((idx, False))
      return moves

    def square_is_empty_or_has_opponents_piece(self, square):
```

(Following lines omitted)

Repository URL: `https://github.com/atifar/checkers/`

An implementation of the game of checkers.

E.2.3 chess

(Preceding lines omitted)

```
 class Pawn(Piece):
   def can_move(self, source, destination):
     if self.color == 'white' and \
         x.index(source[0]) == x.index(destination[0]):
-       ordinary_move = y.index(destination[1]) \
-                       - y.index(source[1]) == 1
+       ordinary_move = abs(y.index(destination[1]) \
+                       - y.index(source[1])) == 1
        two_square_move = not self.moved and \
          y.index(destination[1]) - y.index(source[1]) == 2
        return ordinary_move or two_square_move
```

(Following lines omitted)

Repository URL: `https://github.com/deniscostadsc/chess/`

An implementation of the game of chess.

E.2.4 python-dis3

(Preceding lines omitted)

```
      argval = None
      argrepr = ''
      if op >= HAVE_ARGUMENT:
-       arg = code[i] + code[i+1]*256 + extended_arg
+       arg = code[i] + code[i]*256 + extended_arg
        extended_arg = 0
        i = i+2
        if op == EXTENDED_ARG:
```

(Following lines omitted)

Repository URL: `https://github.com/tomxtobin/python-dis3/`

A back-port of the built-in `dis` module from Python 3 to Python 2.7.

E.2.5 sudoku

(Preceding lines omitted)

```
      }

    def zeroDirect(self, ijk):
-     index, jndex, kndex = ijk
+     index, jndex, jndex = ijk
      root = self.root
      for xndex in self:
        self.bits[xndex][jndex][kndex] = False
```

(Following lines omitted)

Repository URL: `https://github.com/jeffseif/sudoku/`

An engine for solving sudoku puzzles.

Bibliography

[1] Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. *A field guide to genetic programming*. Lulu. com, 2008.

[2] Andries P Engelbrecht. *Computational intelligence: an introduction*. John Wiley & Sons, 2007.

[3] Peter A Whigham et al. Grammatically-based genetic programming. In *Proceedings of the workshop on genetic programming: from theory to real-world applications*, volume 16, pages 33–41, 1995.

[4] Christian Keber and Matthias G Schuster. Option valuation with generalized ant programming. In *GECCO*, pages 74–81, 2002.

[5] Yin Shan, Robert I McKay, Daryl Essam, and Hussein A Abbass. A survey of probabilistic model building genetic programming. In *Scalable Optimization via Probabilistic Modeling*, pages 121–160. Springer, 2006.

[6] Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 207–216. IEEE Press, 2013.

[7] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280. ACM, 2014.

[8] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. De-anonymizing programmers via code stylometry. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 255–270, 2015.

[9] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the*

31st International Conference on Software Engineering, pages 364–374. IEEE Computer Society, 2009.

[10] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 947–954. ACM, 2009.

[11] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for 8 dollars each. In *34th International Conference on Software Engineering (ICSE)*, pages 3–13. IEEE, 2012.

[12] Alistair James Hutton. *An empirical investigation of issues relating to software immigrants*. PhD thesis, University of Glasgow, 2008.

[13] Daniel Jurafsky and James H Martin. Speech and language processing. *International Edition*, 710, 2000.

[14] Fei Song and W Bruce Croft. A general language model for information retrieval. In *Proceedings of the eighth international conference on Information and knowledge management*, pages 316–321. ACM, 1999.

[15] George James Lidstone. Note on the general case of the bayes-laplace formula for inductive or a posteriori probabilities. *Transactions of the Faculty of Actuaries*, 8 (182-192):13, 1920.

[16] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012.

[17] John R Koza. Hierarchical genetic algorithms operating on populations of computer programs. In *IJCAI*, pages 768–774. Citeseer, 1989.

[18] John R Koza. *Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems*. Stanford University, Department of Computer Science, 1990.

[19] David J Montana. Strongly typed genetic programming. *Evolutionary computation*, 3(2):199–230, 1995.

[20] Ivan Tanev. Implications of incorporating learning probabilistic context-sensitive grammar in genetic programming on evolvability of adaptive locomotion gaits of snakebot. In *Proceedings of GECCO 2004*, 2004.

[21] Alain Ratle and Michèle Sebag. Avoiding the bloat with stochastic grammar-based genetic programming. In *International Conference on Artificial Evolution (Evolution Artificielle)*, pages 255–266. Springer, 2001.

[22] Peter A Whigham. Inductive bias and genetic programming. In *Genetic Algorithms in Engineering Systems: Innovations and Applications, 1995. GALESIA. First International Conference on (Conf. Publ. No. 414)*, pages 461–466. IET, 1995.

[23] Nicholas Freitag McPhee and Justin Darwin Miller. Accurate replication in genetic programming. In *ICGA*, pages 303–309, 1995.

[24] Riccardo Poli. A simple but theoretically-motivated method to control bloat in genetic programming. In *European Conference on Genetic Programming*, pages 204–217. Springer, 2003.

[25] Riccardo Poli and Nicholas Freitag McPhee. Parsimony pressure made easy. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1267–1274. ACM, 2008.

[26] Kenneth E Kinnear. Evolving a sort: Lessons in genetic programming. In *Neural Networks, 1993., IEEE International Conference on*, pages 881–888. IEEE, 1993.

[27] William B. Langdon. Size fair and homologous tree crossovers for tree genetic programming. *Genetic programming and evolvable machines*, 1(1-2):95–119, 2000.

[28] Github. URL `http://www.github.com`.

[29] PyPI Ranking. PyPI Python modules ranking. URL `http://pypi-ranking.info/alltime`.

[30] J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2010. ISBN 9781608453436.

[31] Takuya Matsuzaki, Yusuke Miyao, and Jun'ichi Tsujii. Probabilistic CFG with latent annotations. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 75–82. Association for Computational Linguistics, 2005.

[32] Chris Paulson and Giorgos Ragkousis. pykriging: A python kriging toolkit, July 2015. URL `http://dx.doi.org/10.5281/zenodo.21389`.

[33] Donald R Jones, Matthias Schonlau, and William J Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13 (4):455–492, 1998.

[34] pytest. URL `http://docs.pytest.org/`.

[35] pysandbox. URL `https://github.com/haypo/pysandbox`.

[36] M. Dorigo and T. Stützle. *Ant Colony Optimization*. Bradford Company, 2004. ISBN 9780262042192.

[37] Christine Solnon. *Ant colony optimization and constraint programming*. Wiley Online Library, 2010.

[38] Matteo Orrú, Ewan Tempero, Michele Marchesi, and Roberto Tonelli. How do python programs use inheritance? a replication study. In *2015 Asia-Pacific Software Engineering Conference (APSEC)*, pages 309–315. IEEE, 2015.

[39] Matteo Orrú, Ewan Tempero, Michele Marchesi, Roberto Tonelli, and Giuseppe Destefanis. A curated benchmark collection of python systems for empirical studies on software engineering. In *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*, page 2. ACM, 2015.

[40] Giuseppe Destefanis, Steve Counsell, Giulio Concas, and Roberto Tonelli. Software metrics in agile software: An empirical study. In *International Conference on Agile Software Development*, pages 157–170. Springer, 2014.

[41] Sebastian Nanz and Carlo A Furia. A comparative study of programming languages in rosetta code. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 778–788. IEEE Press, 2015.

[42] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 155–165. ACM, 2014.

[43] Slav Petrov, Leon Barrett, Romain Thibaux, and Dan Klein. Learning accurate, compact, and interpretable tree annotation. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, pages 433–440. Association for Computational Linguistics, 2006.

[44] Guido Van Rossum. Whats new in Python 3.0. *What's New In Python 3.0 — Python V3. 0.1 Documentation*, 2011.

[45] Rafe Kettler. A guide to Python's magic methods, 2012. URL `https://web.archive.org/web/20160509000713/http://www.rafekettler.com/magicmethods.html`.