A Study of the Uniqueness of Source Code*

Mark Gabel Zhendong Su
Department of Computer Science
University of California at Davis
{mggabel,su}@ucdavis.edu

ABSTRACT

This paper presents the results of the first study of the *uniqueness of source code*. We define the uniqueness of a unit of source code with respect to the entire body of written software, which we approximate with a corpus of 420 million lines of source code. Our high-level methodology consists of examining a collection of 6,000 software projects and measuring the degree to which each project can be 'assembled' solely from portions of this corpus, thus providing a precise measure of 'uniqueness' that we call *syntactic redundancy*. We parameterized our study over a variety of variables, the most important of which being the level of granularity at which we view source code. Our suite of experiments together consumed approximately four months of CPU time, providing quantitative answers to the following questions: at *what* levels of granularity *is* software unique, and at a *given* level of granularity, *how* unique is software? While we believe these questions to be of intrinsic interest, we discuss possible applications to genetic programming and developer productivity tools.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.8 [**Software Engineering**]: Metrics

General Terms

Human Factors, Languages, Measurement

1. INTRODUCTION

Most creative endeavors produce highly unique artifacts. For example, as authors writing a technical paper, we expect this very sentence to have an extremely high probability of being unique; that is, we expect it to be the first use of these words in this exact order in the history of English prose. We do not have the same intuition when we are *programming*, however. For all of its difficulty and subtlety, the fundamentals of programming often seem rote.

There are several reasons why this may be true. For one, software source code necessarily shares many common *syntactic* elements. Programming languages tend to be fully defined by relatively simple formal grammars that specify structured and idiomatic source code. As a simple example, consider the Java language: the grammar dictates that statements be confined to method bodies, which in turn must reside within class declarations, each of which contains strictly defined sequences of tokens that include keywords and required punctuation. Software engineers impose further homogeneity through voluntary conformance to *style conventions*. Examples include naming conventions that restrict the space of available identifiers and design conventions that bound function and statement sizes.

Commonality in software engineering *tasks* may lead to further similarity in source code. For higher-level tasks, this phenomenon is pervasive and is evidenced by the plethora of reusable software libraries, components, and frameworks that engineers have created to minimize redundant development effort. At a lower level of granularity, repetitive and idiomatic code fragments are common: programs written in C-like languages are often full of simple indexed `for` loops over known bounds, for example.

These traits are all evidence of a *propensity for similarity* in software, which—considering the sheer volume of software in existence and the continued growth of software engineering—suggests the possibility of a "singularity" in software engineering's future: a point of convergence at which all necessary software will have been produced. Taken at face value, this proposition borders on futurist and is somewhat absurd: clearly new requirements and domains will drive new software for the foreseeable future. However, examining the question in terms of *granularity* yields much more realistic scenarios.

For example, consider the C programming language and its associated body of software. At one extreme, it is trivially true that every *token* type in the language has been used at least once, and it is likely true that every legal two-token sequence has been written as well. Once we reach the level of expressions or statements, though, the question becomes more subtle and interesting: although the number of *legal* statements in the language is theoretically infinite, the number of *practically useful* statements is much smaller—and potentially finite. An excessively large arithmetic expression is usually written as a sequence of smaller statements of bounded size, for example. With this in mind, it is entirely possible that every useful statement in the C language has already been written. It is less likely that every useful *block* or *function* has been written, though, but a question of degree then arises: what proportion of the set of practical blocks or functions *have* we written? Stated more generally, just how close are we to writing all the source code we need?

*This research was supported in part by NSF CAREER Grant No. 0546844, NSF CyberTrust Grant No. 0627749, NSF CCF Grant No. 0702622, NSF TC Grant No. 0917392, and the US Air Force under grant FA9550-07-1-0532. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

FSE-18, November 7–11, 2010, Santa Fe, New Mexico, USA.

This paper presents the results of the first study that addresses this question. Our work consists of a large scale study of open source software written in three of the most widely used languages: C, C++, and Java. We approximate the "body of all software" with a corpus of 420 million lines of source code. Our high-level methodology consists of examining a collection of 6,000 software projects and measuring the degree to which each project can be "assembled" solely from portions of the corpus, thus providing a precise measure of "uniqueness." We parameterized our study over a variety of variables, forming a suite of experiments that together consumed approximately four months of CPU time. Collectively, our primary contribution is in providing a quantitative answer to the following question: *how unique is software?*

While we believe that the answer to this question is of intrinsic academic interest, there are several practical applications and consequences:

Automation of Programming Recent research [20] has demonstrated notable improvements in the application of *genetic algorithms* to software engineering tasks. *Genetic programming* (GP) involves the use of genetic algorithms to automate the act of programming [10]; a consequence of a lack of uniqueness of source code—at least at certain levels of granularity—is that the enormous search space for these algorithms could be significantly constrained. While the full automation of system-level development tasks is likely to remain elusive, tasks at the levels of granularity at which software tends to lack uniqueness—ideally the most menial and repetitive tasks—may be quite accessible to GP systems.

Development Tool Research Modern development environments often provide a form of *code completion*, which allows a developer to save time by typing a partial symbol name, like `Str`, and receiving and accepting suggestions for the complete name, like `String` and `StringBuilder`. Similarly, predictive text input schemes on mobile devices allow the completion of natural language words, with rich extensions in the form of *phrase completion* forthcoming [16]. A lack of uniqueness in source code would suggest the opportunity for research into analogous extensions for development tools: we may be able to be extend standard code completion to allow for full-statement, code block, or even function completion.

Code Reuse By quantifying the level of code sharing—incidental or intentional—across a large body of software, our work can serve as a sort of limit study on the potential for code reuse. A line of recent research has sought to provide advanced tool support for *small-scale* code reuse [4]; our results provide a concrete measure of these tools' applicability at various levels of granularity and may provide direction for future research.

This work is organized as follows: in the following section (Section 2), we survey related work and discuss this study's novelty. In Section 3, we describe our methodology, which is followed by our results (Section 4), which suggest a significant lack of uniqueness of software at certain levels of granularity. Finally, we discuss various threats to validity (Section 5) and our plans for future work (Section 6).

2. RELATED WORK

While we believe our study of uniqueness to be the first of its kind, other areas of software engineering research are related and share similar concepts.

Clone Detection Clone detection is a research area concerned with detecting and studying the copying and pasting of source code fragments, with the first major work being Baker's `dup` tool [3]. Code clone detection is tied to the idea of intentional copying and pasting—a deliberate action of a developer—and the tools, techniques, and studies are informed by this. Our study considers duplication of any kind, but our focus is on *incidental* similarity (or lack thereof). The effect of intentional code clones on our results is minimal, and we in fact consider them to be a minor nuisance that we explicitly control for (*cf.* Section 4). That aside, two studies of cloning are relevant to this study.

Language	Projects	Source Files	Lines of Code
C	3,958	349,377	195,616,239
C++	1,571	253,754	89,827,589
Java	437	595,271	122,820,171

Table 1: Corpus summary

Language	Tokens/Line (Avg. / Median / P_{75} / P_{95} / P_{99}) All non-blank lines	Lines w/ >1 token
C	6.66 / 5 / 9 / 17 / 29	7.95 / 6 / 10 / 18 / 30
C++	6.95 / 6 / 10 / 19 / 29	8.72 / 7 / 11 / 20 / 32
Java	6.90 / 6 / 9 / 16 / 28	8.48 / 7 / 11 / 17 / 32

Table 2: Relating tokens to lines across the corpus

Liveri *et al.* present a large scale study of code clones within the FreeBSD ports collection, a body of code similar in size to the subjects of our study [13]. They solve the related scalability problem with a parallel version of Kamiya *et al.*'s CCFinder clone detection tool [9]. Their results are dominated by *file-level* code clones, *i.e.* copies of entire files. In our study of uniqueness, we consider these to be an artificial source of similarity that we also explicitly control for.

Al-Ekram *et al.* present a much smaller study [1] of cloning between open source software projects that share similar functionality. They find a general lack of these 'cross-project' clones, but they note a nontrivial amount of incidentally similar code fragments that result from the use of similar APIs. This study hints at one source of a potential lack of uniqueness in software and serves as partial motivation for our study.

Finally, as a somewhat superficial difference, we note that although we consider a variety of levels of granularity, the bulk of our interesting results fall at levels of granularity squarely below those set as minimum thresholds in both past [3] and recent [7] clone detection tools; in effect, we have fully explored the space ignored by these tools.

Studies of Reuse Mockus presents a study of large-scale code reuse [14] in open source software over a massive, continually-growing dataset [15], which is approximately five times larger than ours. However, this study only considers *file-level* reuse, which we explicitly control for and ignore, rendering it complementary to our own. Research into reuse metrics [5] seeks to quantify the time and resources saved by intentional reuse, while our study focuses on incidental similarity, which may indicate the *potential* for reuse.

Most recently, Jiang and Su present a new tool, EqMiner [8], that locates functionally identical code fragments based completely on testing. The authors' motivation for the study is to explore the *functional* uniqueness of programs, while ours is to explore *syntactic* uniqueness. This semantics-based study is complementary to our own, and extending it to our scale would be especially interesting.

Code Search Code search platforms often operate over collections of software at or in excess of the size of our own study [2, 18], with less scalable variants allowing for more flexibility through semantic search using test cases [12, 17]. This line of research can benefit from our study by treating it as a form of limit study on various types of syntactic reuse: levels of granularity at which software is

highly unique would form ideal candidates for more advanced and flexible search techniques.

Schleimer *et al.*'s MOSS [19] uses a form of document fingerprinting to scalably characterize likely-plagiarized software. This work is based on the assumption of a certain level of uniqueness in software; our study directly measures this value for a large sampling of software, and our results may possibly lead to more accurate and complete plagiarism tools.

3. STUDY DESIGN

Our study aims to answer the general question of the *degree of uniqueness* of software. While simple conceptually, this question is fraught with subtlety and complexity. This section describes our methodology in detail, with special attention given to the rationale for our various design choices that may affect the validity of our results. We start with a high-level summary of our methodology (Section 3.1) and continue with a discussion of our experimental variables (Sections 3.2–3.5). We then summarize with a complete inventory of our suite of experiments (Section 3.6).

3.1 High-level Methodology

This study is based on a metric for uniqueness that we define in terms of a software *project*. We begin our description with a simple thought experiment that illustrates our intuition:

> You are a software engineer starting a new project, requirements in hand. Unfortunately, your keyboard has malfunctioned, and your *only* method for entering program text is through copying and pasting *existing* code fragments. Fortunately (perhaps), you have oracle-like access to *all* source code that has ever been written. How much can you accomplish?

This amount—the proportion of the project that you are able to complete—is the essence of our metric. A low value for a specific project indicates that the project is unique; low values for *all* projects would indicate that *software* is unique.

When defined as the object of a thought exercise, this metric is inherently impossible to calculate. We take three steps in concretizing it as a computable value. First, we approximate 'all code that has ever been written' with a large collection of source code, which we call the *corpus*. Next, we precisely define our representation—our 'view'—of source code, which then leads to a natural methodology for calculating uniqueness at various levels of granularity.

Gathering a Corpus Our corpus consists of a collection of open source software in three languages: C, C++, and Java. We collected the bulk of our corpus from the complete source distribution of the current release (12) of the Fedora Linux distribution. A Linux distribution like Fedora has several properties that benefit our study:

Size: With a collection of software designed to accommodate the needs of a large user base, a Linux distribution contains a vast amount of source code.

Diversity: The large amount of source code is distributed among a proportionally large collection of software projects with diverse functionality.

Lack of Duplication: Fedora is distributed as a managed system of packages and is likely to contain little large-scale duplication, preferring package-based dependencies in place of copies. While copies of source code in the corpus would not taint our results, a study of this scale must use CPU time economically, and scanning duplicates is an obvious waste.

This collection contains an adequate amount of C and C++ code, but we found Java to be underrepresented. To complete our corpus, we supplemented it with a similarly diverse collection of Java projects from `http://www.java-source.net`. A summary of our complete corpus appears in Table 1.

Language	Project	Description	Size (Lines)
C	atlas	Linear algebra kernel	554,342
	ffdshow	Audio/video codec	361,164
	freedroid	Arcade game	75,926
	grisbi	Personal accounting	162,033
	net-snmp	SNMP library	306,598
	pnotes	Desktop notes	29,524
	sdcc	Small device C compiler	138,093
	tcl	Scripting language	226,499
	winscp	SCP client	174,936
	xbmc	Media center	1,947,596
C++	7zip	Compression tool	104,804
	audacity	Audio editor	209,073
	dvdstyler	DVD video authoring	17,764
	hugin	Panoramic photo stitcher	91,234
	mediainfo	Media file identifier	100,312
	mumble	Voice communication	73,858
	npp	Text editor	91,515
	ogre	3d engine	392,212
	postbooks	ERP/CRM and accounting	271,377
	scummvm	Adventure game engine	738,262
Java	adempiere	ERP/CRM business suite	1,174,343
	arianne	Multiplayer game engine	198,539
	freecol	Strategy game	164,797
	jedit	Development environment	176,508
	jmri	Model railroad interface	354,431
	jtds	JDBC driver	66,318
	openbravo	Web-based ERP	192,020
	rssowl	Feed reader	169,077
	sweethome3d	Interior design tool	73,180
	zk	Ajax framework	181,207

Table 3: Target projects retrieved from SourceForge

Source Representation This study takes a *lexical* view of source code, representing each source file as a simple sequence of *tokens*. We found this to be a viable compromise between scalability and flexibility. A simpler, line-based view would be inexpensive to compute but highly brittle; a measure of uniqueness at the line level is liable to be an overestimate. A more rich representation, like syntax trees or dependence graphs, would allow for more flexibility but would be expensive to compute for 6,000 projects. In addition, overly abstract representations are increasingly disconnected from the actual act of programming, which we would like our study to relate to as much as possible.

To more conveniently relate our lexically-based study to more familiar measures, we performed a brief study of the distribution of tokens over lines on our corpus, the results of which appear in Table 2. The distributions are quite consistent across the three languages, likely owing in no small part to style conventions. For each language, we report statistics for both 1) all non-blank, non-comment lines and 2) lines with more than one token.

Methodology Recall that our metric for uniqueness is conceptually based on the amount of a given project that can be 'assembled' from already-written code. With a corpus and code abstraction selected, we concretize this concept in a measure we call *syntactic redundancy*. We calculate this metric for a specific software *project* with respect to a *corpus* and a given *level of granularity*, which, under our lexical view, we define in terms of contiguous fixed-length token subsequences (token-level *n*-grams, or *shingles*).

Briefly, a given token in the input project is 'syntactically redundant' if it is enclosed in at least one sequence that is 'matched' by some sequence in the corpus. *Syntactic redundancy* is the proportion

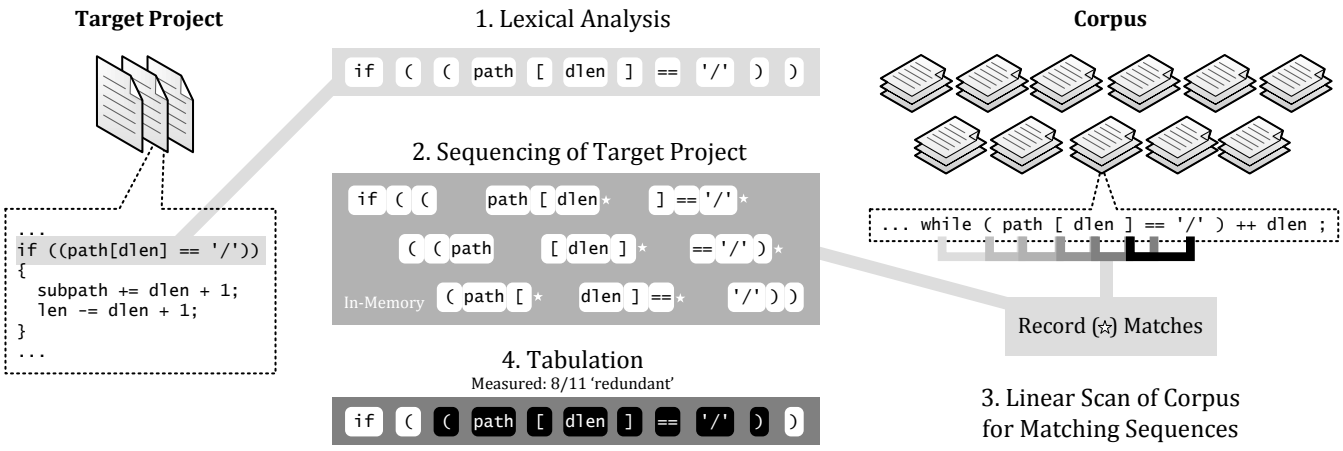


Figure 1: Overview and example of our methodology for calculating 'syntactic redundancy'

of 'redundant' tokens in a given project. We illustrate this process graphically in Figure 1 and narrate the steps here.

Input As input, we have 1) a target *project* and 2) a *corpus*, which are both collections of program source files, and 3) a positive integer g representing the *level of granularity*.

i. Sequence the Target Project First, we analyze each of the target project's source files according to the lexical specification of the target language. We then enumerate every fixed-length token subsequence of size g (*i.e.*, the token-level g-grams) and load them into memory—a process we call 'sequencing.'

ii. Scan, Sequence, and Process the Corpus We then perform a linear scan of the corpus, sequencing each corpus file under the same conditions. For each corpus sequence, we search for a *match* in the target project. If found, we record the tokens in the target project that comprise the matching sequence as 'redundant,' reflecting the fact that we could have copied and pasted this corpus sequence to form this segment of the target project. Note that due to overlap, tokens may be marked as redundant multiple times, but each token will only be counted once.

iii. Collect Results The process concludes with a final tabulation of the target project's sequences (which are processed and residing in memory), returning the ratio of 'redundant tokens' to 'all tokens' as a percentage.

With our basic methodology in place, we continue with a description of our four main experimental variables.

3.2 First Variable: Target Projects

The first variable in our experiments is the *target project*. We chose two sets of target projects in an effort to compromise between depth and breadth of study: our first (small) set of projects allows us to perform a large number of measurements with absolute precision, while our second (*much* larger) set allows us to more accurately draw general conclusions about the uniqueness of software.

First Set: Depth We retrieved our first set of targets from SourceForge[1] by walking the 'Most Active Projects – Last Week' list and collecting the top 10 primarily-C, C++, and Java projects. Descriptions of these 30 projects appear in Table 3.

[1] `http://sourceforge.net`

Selecting from list has several advantages. First, the ranking is in terms of site *activity*—including bug tracker activity, forum posts, and releases—which prevents us from considering abandoned projects. Second, the scope of the ranking criteria is limited to a window of a single week, providing an opportunity for both new and mature projects to appear. Third, the list *excludes* downloads as a criterion; lists that include downloads are dominated by peer-to-peer file sharing applications and would not have resulted in a diverse study.

Second Set: Breadth We increase the breadth of our study with a second set of projects: the 6,000 corpus projects themselves. Calculating syntactic redundancy for each project in this set would ordinarily be prohibitively expensive: directly applying our technique from Section 3.1 would amount to a quadratic-time scan of 420 million lines of code.

Fortunately, our methodology lends itself to stochastic estimation. Our estimation technique is illustrated in Figure 2 and is largely straightforward. We treat the *syntactic redundancy* of a token as a binary random variable and estimate its value by sampling uniformly from the population of tokens in a project. We determine the number of necessary samples according to a desired margin of error, confidence, and project size using standard techniques [11]. In our experiments over these 6,000 projects, we calculated redundancy with a $\pm 2.5\%$ margin of error and 95% confidence.[2]

There is one subtlety, which is reflected in Figure 2: to obtain correct results, we must correctly compute the redundancy of each sampled token. This involves generating *all* sequences that include each sampled token, even if they include other, *non*-sampled tokens. However, these non-sampled tokens are not to be counted when tabulating, as they do not belong to the 'sampled' project. Note also that sampling is limited to the target project: after sampling, we always perform a scan of the *entire* corpus.

Though we believe our methodology to be sound, we did validate it empirically on our 30 SourceForge projects by comparing the precisely and approximately computed values. We found the estimation to be quite accurate in all trials, within a $\pm 1.0\%$ range of the true value despite using the above parameters.

3.3 Second Variable: Granularity

Our second experimental variable is the level of granularity, g, which controls the length of the token sequences our analysis gener-

[2] In practice, enforcing these parameters requires sampling approximately 1,500 tokens per project. Variations are due to differing project sizes.

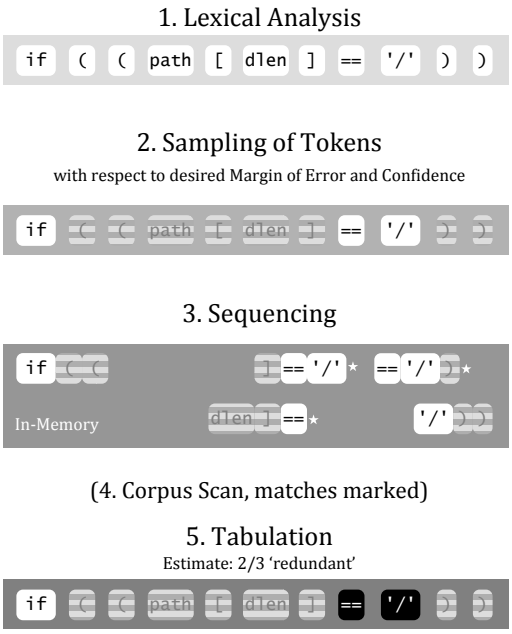


Figure 2: Scalable estimation technique for our breadth experiments

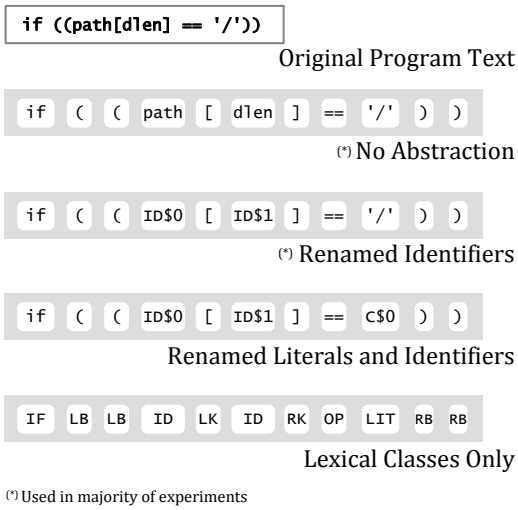


Figure 3: Possible levels of abstraction for token sequences

ates during the 'sequencing' phase. Put more plainly in terms of our earlier thought exercise, g represents the size of the code segments that we copy from the existing code to 'assemble' a project. We parameterize our experiments over a variety of token sequence lengths. Our goal is to view our study from two complementary perspectives: at *what* levels of granularity *is* software unique, and at a *given* level of granularity, *how* unique is software?

We set our values for granularity by first selecting two critical values, a functional minimum (the median value of tokens in a single line) and maximum (determined through exploratory experiments), that denote a range of relevant values. The scale of our study prevents us from performing experiments on every value in this range; we instead select g through a series of 'passes' over this range: in the first pass, we perform experiments over a small, fixed number of values; in subsequent passes—the number of which are limited by our computing resources—we iteratively refine our information by performing experiments on the *midpoints* of previously-computed values of g.

3.4 Third Variable: Matching Criteria

In our methodology, we mark a token as *redundant* when we find a 'matching' sequence for it in the corpus. While the most clear and obvious definition of 'match'—exact equality—is the most intuitive, we also explored alternate definitions that allow for some imprecision.

It may not be clear at first why approximate matches are worth exploring: on the surface, such an extension might seem to do little more than inflate the results. Note, though, that we are only interested in specific cases involving *slight* imprecision: it may be the case that a trivial 'nudge' causes the threshold for uniqueness in software to be significantly higher, and it is precisely this idea—the interplay between granularity and uniqueness—that we wish to explore fully. Our intuition here is experiential: as software engineers, we hypothesize that many potential matchings may fail due to very slight differences, like an operator in an expression, the name of an identifier, or the ordering of two parameters.

We expand our experimental infrastructure with the ability to detect matches within a specified *Hamming distance*. Briefly, Hamming distance is a metric defined over two sequences of equal length, and it is calculated as the number of positions at which their elements differ.

Implementing this extension in a scalable manner is certainly one of this study's more involved engineering tasks. While finding all *exactly* matching sequences is fast and straightforward using hashing, the naïve approach to finding all *approximate* matches involves a linear number of expensive Hamming distance calculations per query. We solve this problem with a new randomized hashing algorithm that is related to the concept of Locality Sensitive Hashing [6]. The technical details and proofs are outside the scope of this paper; we focus instead on the theoretical properties that affect the validity of our results:

Precision As a randomized—not an approximation—algorithm, our algorithm only returns *true* matches; that is, it returns no false positives.

Recall The probability of missing an arbitrary match has provable bounds and is tunable. We set this probability to 0.99 in our experiments.

As applied to this study, these properties can be summarized plainly: any syntactic redundancy value we report (that allows for imprecision) is a *sound* underapproximation that is 'very likely' near the true value.

In our experiments, we measure syntactic redundancy with respect to a) exact matches only and b) matches allowing for a Hamming distance of up to 1, 2, 3, and 4 (separately reported).

3.5 Fourth Variable: Abstraction

In our example depicted in Figure 1, the sequences consist entirely of concrete program text: the only processing is in separating the tokens according to a lexical specification. In programming, however, some aspects of concrete program text are arbitrary and do not affect semantics, with a simple example being *variable naming*. Failing to account for this may result in our study reporting that software is overly unique.

We can control for this kind of variation by *abstracting* the individual token sequences from their concrete program text to a more normalized form. Various schemes are possible; we present four

methods in Figure 3. In the bulk of our experiments, we limit ourselves to the two most conservative types: *none*, or no abstraction at all, and *renamed identifiers*, which involves a consistent alpha renaming of all 'identifier'-typed tokens but leaves all other tokens—including the values of literals—untouched.

3.6 Summary

We have defined a general methodology and four experimental variables. Our final set of experiments includes the calculation of syntactic redundancy under the product of the following conditions:

1. Two sets of target projects: 30 SourceForge projects under full precision (depth) and 6,000 corpus projects under stochastic estimation (breadth).
2. Two levels of abstraction: *none* and *renamed identifiers*.
3. Exact 'matches' and matches allowing for maximum Hamming distances of 1, 2, 3, and 4.
4. As many levels of granularity as our computing resources will allow.

4. RESULTS

This section presents the results of our study. We begin with a brief description of the operation of our experiments, including an overview of our implementation and the number and type of experiments we completed. We then present the results of our 'depth' experiments over the 30 SourceForge projects, which is followed by a discussion of the results of our 'breadth' experiments over all 6,000 corpus projects. In the following sections, our emphasis is on presenting our data in as raw, clear, and unbiased a form as possible.

4.1 Implementation and Operation

Our experimental infrastructure consists of an entirely new, highly optimized Java application running on a dual-Xeon server with 16 GB of main memory. Notable features include:

- The ability to calculate syntactic redundancy for multiple projects in parallel, which was critical for our breadth experiments.
- Sound and efficient computation of approximate matches (*cf.* Section 3.4). In addition, we can compute redundancy for multiple values of 'maximum Hamming distance' simultaneously, with little marginal cost in both time and space.
- Simple extension to other languages by providing a lexical specification.
- Resource awareness: our application expands to consume all available memory and CPUs as efficiently as possible.

As a 100% Java application, our infrastructure should run on any Java-supported platform. In practice, however, we require a 64-bit system and virtual machine, and certain optimizations present in Sun's 1.6+ reference implementation are essential to making reasonable progress at this scale.

As we described in the previous section, we fix a set of values for all but one experimental variable, granularity (g, expressed in tokens); we instead perform what amounts to a systematic 'search' over various values between a functional minimum and maximum. At the time of this writing, our experiments have consumed a total of approximately four months of CPU time and have completed redundancy measurements for the following levels of granularity:

Depth: 6, 9, 13, 16, 20, 23, 27, 31, 35, 56, 77, 98, 120

Breadth: 6, 20, 35, 77, 120

These cumulative results have provided a sufficient quantitative answer to our question of the uniqueness of software.

4.2 Depth: SourceForge Projects

The results of our experiments over the 30 SourceForce projects appear as plots in Figure 4. The independent variable is the level of granularity in tokens, g, and the dependent variable is the syntactic redundancy of the project expressed as a percentage. Each line on each plot represents the redundancy value with respect to two variables: abstraction (Section 3.5) and matching criteria (Section 3.4).

We perform no estimation or approximation during these 'depth' experiments, and we present the results in as raw a form as possible. We briefly summarize the extent our filtering and summarization: a) for clarity of presentation, we omit lines for 'Max Hamming distance ≤ 4,' though the data are available; b) due to the dramatic drop off in redundancy after $g = 50$, we focus each graph on $g \leq 65$; and c) the resolution of these graphs may give the illusion of smoothing on many of the lines, but we did not perform any.

The most striking feature of these graphs is their similarity: apart from a few outliers, all appear to follow a similar trend. First, at the minimum level of granularity, 6 tokens (or approximately one line), between 50% and 100% of each project is redundant, depending on matching and abstraction. At this point, the sets of lines take two paths: the 'no abstraction' set drops off and flattens quickly, while the 'renamed identifiers' set maintains a period of flatness at which redundancy is high. After a more delayed decline, they reach an inflection point at around 20-25 tokens, flatten out, and finally join the 'no abstraction' lines. In all but one experiment, we measured no significant amount of redundancy at levels of g over 60.

The convergence of all redundancy values to a common level highlights an ancillary benefit of our choice of abstraction levels: *control for clones* and the resulting *focus on incidental similarity*. At sufficiently high levels of granularity, the 'no abstraction' results can be interpreted as being generally composed of intentional copying, while the 'renamed identifiers' results are more likely to include incidental similarity. (Note that this characterization is not perfect, only probable: code can be incidentally completely identical, and intentional copies can be consistently renamed or otherwise adapted.) The extensive *spread* between the two sets of values between $g = 10$ and $g = 40$—consistent across all projects—suggests a substantial amount of incidental similarity. When the two sets of measurements finally meet at higher levels of granularity, they have converged on the (comparatively rare) intentional copies—the code clones.

Controls for Trivial Redundancy The root causes of redundancy are important; trivial cases like full file copies and intentional code clones are uninteresting, as our goal is to study the intuitive idea of *incidental* similarity in source code. In addition to the information provided by the two abstraction levels, we implemented a small assortment of other controls for trivial redundancy. Though more properly described with our methodology, we believe the current context provides more intuition. We developed these controls after earlier exploratory experiments with a (much less scalable) version of our platform that provided full traceability information for every match.

First, for the Java language, we do not measure the initial segment of every source file that contains `import` statements, which control namespace resolution. Similarly, for C and C++, we ignore the standard string of `#include` directives and `using` statements at the start of each file, and we ignore all header files, which usually only contain declarations. Our focus is on studying the semantics-affecting, intentional aspects of programming, and these controls allow us shift focus from the most egregiously spurious, structurally induced matches.

Second, for all languages, we do not allow matches from duplicated files (determined by content) *or* files with identical file names. The latter—somewhat aggressive—filter allows us to conservatively

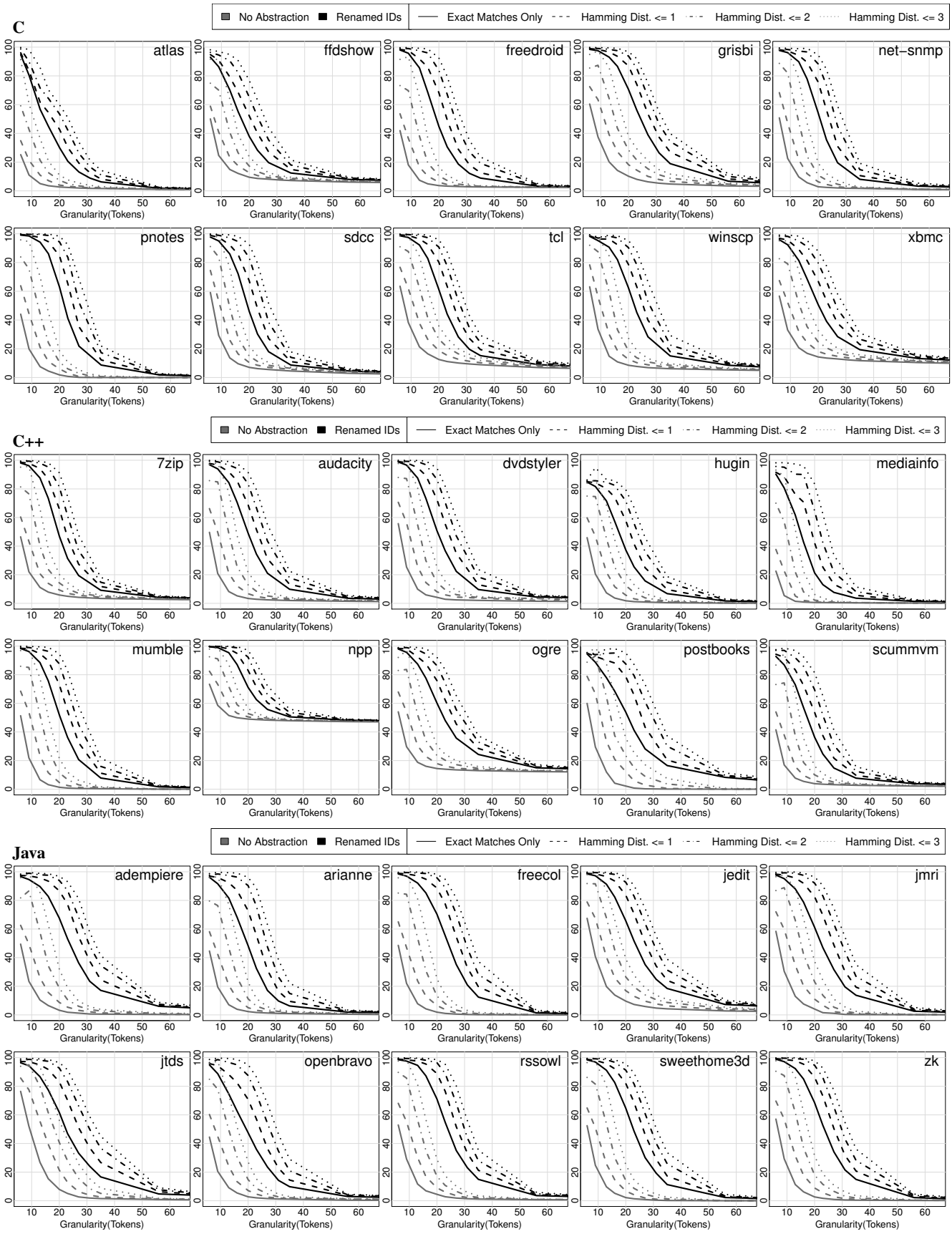


Figure 4: Syntactic Redundancy of Sourceforge Projects. The dependent variable is the percentage of syntactically redundant tokens.

control for the cases in which copied files are very slightly adapted (*e.g.* a copyright header) or are of different but similar versions. Before adding this filter, we did measure its potential effect: in all cases, it reduces the relative size of the corpus (with respect to a single project) by less than one percent.

Outliers The plot of `npp` follows the standard trend, but it converges on a much higher value of redundancy. Upon investigation, we noted that over half of the project's source code consists of generated lexers used for syntax highlighting, which at least one project in the corpus undoubtedly contains as well. Other interesting projects included `atlas` and `hugin`, with the former showing an abnormally steep drop off in redundancy and the latter having an exceptionally low redundancy value at low levels of granularity. We do not have a complete explanation for these phenomena, but we hypothesize that they are a result of the projects' specialized domains (linear algebra solving and panoramic photo stitching, respectively).

One minor effect is exhibited in a minority of the graphs: at very low levels of granularity, slightly increasing granularity counterintuitively *increases* redundancy. This is an artifact of our methodology: at a given level of granularity g, only files with at least one g-sized sequence are counted as part of the project. We could have formulated our measurements either way, either including or excluding tiny files, but in any case, the effect is negligible: these files generally only contain between 0–1.5% of any given project's code.

In summary, we observe a substantial amount of incidental similarity between these projects and our corpus. The bulk of the syntactic redundancy is exhibited at significant, but still fairly low, levels of granularity: 6 to 40 tokens, or approximately one to seven lines of code.

4.3 Breadth: Corpus Projects

Our breadth experiments involve calculating estimated syntactic redundancy values for all 6,000 of our corpus projects. Summary statistics of the results appear in Table 4, and we have included density plots of the distributions of these values, overlaid for each language, in Figure 5. In this section, we restrict the language of our observations to general and qualified terms: other than basic summarization, these data are 'raw,' and we have not formally formulated/tested any hypotheses, and we have not performed any statistical tests. Here, our primary contribution is in the *collection* of a vast amount of previously unknown (or perhaps unattainable) data; our interpretations are secondary and are *suggestions* of general trends.

At $g = 6$, approximately one line of code (shown Figure 5a), the projects are nearly wholly redundant when measured under abstraction, and their values of syntactic redundancy are over half when measured using no abstraction at all. All three languages are apparently in agreement, which suggests the possibility that individual lines of code are *not* unique.

The next level of granularity, $g = 20$ (Figure 5b) is more interesting. In our depth experiments, this level of granularity falls in the center of the range of values at which we observe a high redundancy values. On the whole, these aggregate redundancy measures are essentially in agreement with the individual values for our SourceForge experiments, but the individual languages are less in agreement with each other: the suggestion of a general trend is still there, but we observe more variation. The Java projects, for example, appear to have a generally higher level of redundancy, while the C and C++ measurements are much closer to each other in value. Once again, we observe a substantial and consistent *spread* between the abstracted and non-abstracted measurements, suggesting a general trend of *incidental* similarity.

At $g = 35$ (Figure 5c), our observations are again in line with our depth experiments: we observe generally more uniqueness (*i.e.*, less redundancy), and the spread between the abstracted and non-abstracted measurements significantly narrows. At $g = 77$ (Figure 5d) and 120 (no figure, but displayed in Table 4), we observe near-total uniqueness, and we also observe a potential broad-scale confirmation of the phenomenon of the redundancy measures *converging* on the more rare, intentionally copied code fragments: both the abstracted and non-abstracted distributions appear centered around similar values.

Across all runs, our measurements are in agreement with our depth experiments: redundancy is near total at the line level and remains significant through the range of approximately one to six lines.

			Median Syntactic Redundancy (%)				
			Max Hamming Dist:				
	g	**Abstraction**	0	1	2	3	4
C	6	None	63.3	74.8	88.4	96.7	99.9
		Renamed IDs	98.3	98.7	99.0	99.6	99.9
	20	None	7.8	14.0	23.6	34.8	49.9
		Renamed IDs	59.5	79.6	90.8	96.1	98.5
	35	None	4.1	5.5	7.2	9.1	11.1
		Renamed IDs	14.8	19.5	25.0	30.8	37.3
	77	None	2.0	2.4	2.7	3.1	3.4
		Renamed IDs	4.5	5.0	5.6	6.0	6.5
	120	None	1.4	1.6	1.8	1.9	2.0
		Renamed IDs	2.7	2.9	3.1	3.2	3.4
C++	6	None	54.5	68.9	84.8	95.8	99.8
		Renamed IDs	97.9	98.5	99.2	99.8	100.0
	20	None	3.2	7.8	15.1	25.2	39.3
		Renamed IDs	48.1	68.2	83.6	92.4	96.9
	35	None	0.9	1.5	2.4	3.6	5.3
		Renamed IDs	9.8	13.3	18.0	22.4	27.8
	77	None	0.1	0.3	0.3	0.5	0.6
		Renamed IDs	1.6	1.8	2.1	2.3	2.6
	120	None	0.0	0.0	0.1	0.1	0.1
		Renamed IDs	0.7	0.8	0.9	0.9	1.0
Java	6	None	69.5	81.0	92.9	98.5	99.9
		Renamed IDs	98.2	98.5	98.8	99.5	99.9
	20	None	9.6	18.1	30.5	45.9	63.5
		Renamed IDs	72.2	88.1	95.4	98.1	99.2
	35	None	3.9	5.6	8.0	10.8	14.1
		Renamed IDs	23.0	30.4	39.7	48.5	56.5
	77	None	1.8	2.2	2.6	2.9	3.3
		Renamed IDs	4.9	5.3	5.9	6.4	7.0
	120	None	1.3	1.5	1.7	1.8	1.9
		Renamed IDs	2.6	2.9	3.1	3.3	3.5

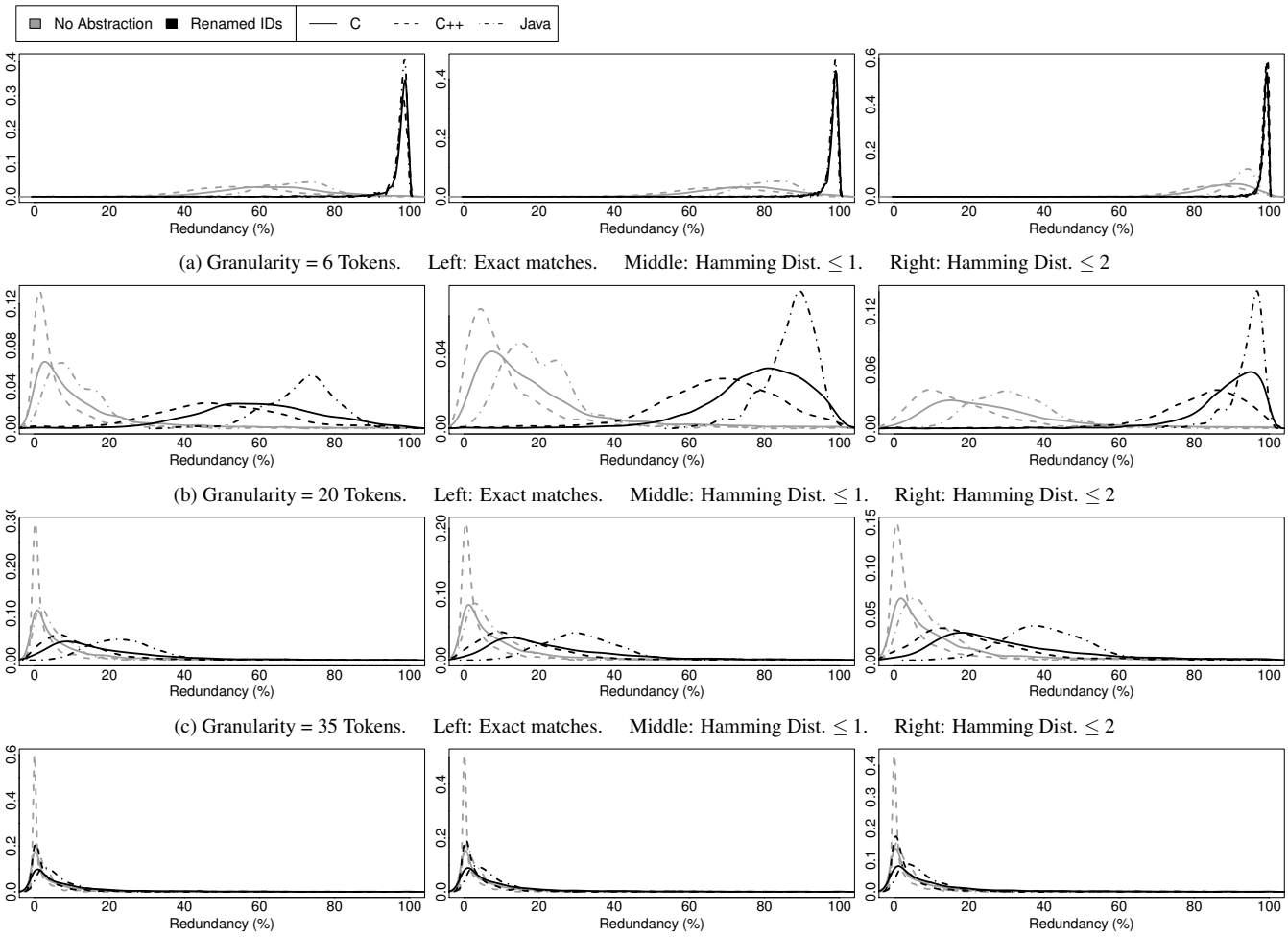
Table 4: Median syntactic redundancy values for the 6,000 corpus projects.

5. THREATS TO VALIDITY

Threats to the validity of our study fall under two main categories: construct validity and external validity.

Construct Validity The construct validity of our study rests on our ability to accurately measure 'true' syntactic redundancy, a measure that we have approximated concretely in terms of a *corpus* in the hope that it provides an accurate estimation of the same value computed for 'all code in existence.'

Here, the most obvious threat is that our corpus is insufficiently large or varied, leading us to potentially *under*report redundancy. We believe this to be unlikely: the corpus is highly diverse, and we report quite similar measurements for all three languages, despite the fact that the majority of the Java and C/C++ portions of corpus are derived from wholly different sources.



(a) Granularity = 6 Tokens. Left: Exact matches. Middle: Hamming Dist. ≤ 1. Right: Hamming Dist. ≤ 2

(b) Granularity = 20 Tokens. Left: Exact matches. Middle: Hamming Dist. ≤ 1. Right: Hamming Dist. ≤ 2

(c) Granularity = 35 Tokens. Left: Exact matches. Middle: Hamming Dist. ≤ 1. Right: Hamming Dist. ≤ 2

(d) Granularity = 77 Tokens. Left: Exact matches. Middle: Hamming Dist. ≤ 1. Right: Hamming Dist. ≤ 2

Figure 5: Density plots of the distribution of syntactic redundancy values for 6,000 corpus projects.

We also performed a small experiment to provide (limited) evidence of the sufficiency of our corpus, the results of which appear in Figure 6. In this experiment, we fix all our standard experimental variables: a single project (freecol), a single level of granularity (20) and abstraction (renamed identifiers), and, for simplicity, exact matches only. We instead vary the size of the *corpus* through sampling: we first compute syntactic redundancy with respect to a sampled corpus of approximately 1,000 files. We then repeatedly double the size of the corpus and remeasure until we reach the original size, 600,000 files. As expected, the syntactic redundancy increases monotonically, but the growth rate is highly variable, strong at first but trailing off dramatically before even half the corpus has been sampled. Though only a single data point, this experiment suggests that increasing the scope of our corpus may not yield substantially different measurements.

There are also potential threats due to errors in our implementation. We did utilize end-to-end regression testing throughout our platform's development and optimization, however, and we are confident our measurements. In addition, we are willing to release our implementation on request (and all raw data, for that matter) for inspection and/or replication.

External Validity The general trends apparent in our depth experiments may not generalize to most or all software. Our breadth experiments over 6,000 projects do help in confirming a general trend, but there is a potential threat from platform homogeneity: our corpus, which comprised the set of *target* projects in the 'breadth' experiments, is composed completely of open source Linux software. However, we believe that this threat is mitigated by the abundance of cross-platform software and the fact that only a small fraction of code in high-level languages is likely to be truly platform-specific. In addition, a selection of our SourceForge projects are actually *Windows-only* projects, and our measurements for these projects are consistent with the rest despite being measured against a Linux-based corpus.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we present the first study of the *uniqueness* of source code. We have formulated a precise and intuitive methodology for measuring this value, which we call *syntactic redundancy*. We compute this value precisely for 30 assorted SourceForge projects and approximately—but with known error margins and confidence—for 6,000 other projects. Our experiments, covering 430 million lines of source code and consuming approximately four months of CPU time, revealed a general lack of uniqueness in software at levels of granularity equivalent to approximately one to seven lines of source code. This phenomenon appears to be pervasive, crossing both project and programming language boundaries.

Our most immediate line of future work is the exploration of the practical ramifications of our findings. We are most interested in the consequences of this study on *genetic programming*, which we

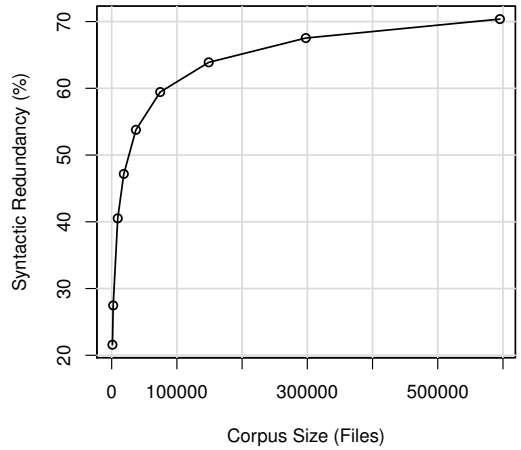


Figure 6: The syntactic redundancy of `freecol` with respect to sampled subsets of our corpus.

remarked on earlier: a fitness function that ensures that a proposed program 'looks like software should look' could greatly improve the performance of these systems, possibly making many problems tractable for the first time.

We are also interested in repeating our study with different targets and/or corpora. When retrieving the 'depth' targets from SourceForge, we were encouraged by the fact that our chosen languages—C, C++, and Java—dominated the list of the most active projects. However, web languages—PHP in particular—came in a close fourth. Repeating our study for this and other non-C-like languages may yield quite different results. We would also like to scan and tabulate commercial code as well, though we doubt the results would be significantly different.

A natural complement to our quantitative line of work is a thorough investigation into the *qualitative* aspects of syntactic redundancy. For example, it may be the case that a particular set of common sequences—software 'genes'—dominate the results, which could drive tool support. Earlier versions of our measurement infrastructure did allow for the full tracing of every match, but we ultimately had to drop this feature in favor of scalability. As future work, we intend to explore methods of reinstating this feature without compromising our ability to scan large amounts of source code.

7. REFERENCES

[1] R. Al-Ekram, C. Kapser, R. C. Holt, and M. W. Godfrey. Cloning by accident: an empirical study of source code cloning across software systems. In *ISESE*, pages 376–385, 2005.

[2] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 681–682, 2006.

[3] B. S. Baker. On finding duplication and near-duplication in large software systems. In *WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering*, page 86, 1995.

[4] R. Cottrell, R. J. Walker, and J. Denzinger. Semi-automating small-scale source code reuse via structural correspondence. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 214–225, 2008.

[5] P. Devanbu, S. Karstu, W. Melo, and W. Thomas. Analytical and empirical evaluation of software reuse metrics. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 189–199, 1996.

[6] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 518–529, 1999.

[7] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proceedings of ICSE*, pages 96–105, 2007.

[8] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of ISSTA*, pages 81–92, 2009.

[9] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[10] J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, USA, 1992.

[11] R. V. Krejcie and D. W. Morgan. Determining sample size for research activities. *Educational and psychological measurement*, 30:607–610, 1970.

[12] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, R. S. Morla, P. C. Masiero, P. Baldi, and C. V. Lopes. CodeGenie: using test-cases to search and reuse source code. In *Proceedings of ASE*, 2007.

[13] S. Livieri, Y. Higo, M. Matushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder. In *Proceedings of ICSE*, pages 106–115, 2007.

[14] A. Mockus. Large-scale code reuse in open source software. In *FLOSS '07: Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*, page 7, 2007.

[15] A. Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In *MSR '09: Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 11–20, 2009.

[16] A. Nandi and H. V. Jagadish. Effective phrase prediction. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 219–230, 2007.

[17] A. Podgurski and L. Pierce. Retrieving reusable software by sampling behavior. *ACM Trans. Softw. Eng. Methodol.*, 2(3):286–303, 1993.

[18] S. P. Reiss. Semantics-based code search. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 243–253, 2009.

[19] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85, 2003.

[20] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 364–374, 2009.