Using n-grams to rapidly characterise the evolution of software code

Austen Rainer, Peter C.R. Lane, James A. Malcolm, Sven-Bodo Scholz
School of Computer Science, University of Hertfordshire
College Lane Campus, Hertfordshire, AL10 9AB, U.K.
{a.w.rainer; p.c.lane; j.a.malcolm; s.scholz}@herts.ac.uk

Abstract

Text-based approaches to the analysis of software evolution are attractive because of the fine-grained, token-level comparisons they can generate. The use of such approaches has, however, been constrained by the lack of an efficient implementation. In this paper we demonstrate the ability of Ferret, which uses n-grams of 3 tokens, to characterise the evolution of software code. Ferret's implementation operates in almost linear time and is at least an order of magnitude faster than the diff tool. Ferret's output can be analysed to reveal several characteristics of software evolution, such as: the lifecycle of a single file, the degree of change between two files, and possible regression. In addition, the similarity scores produced by Ferret can be aggregated to measure larger parts of the system being analysed.

1. Introduction

In simple terms, software evolution investigates the changes that occur to a software system's source code over time. As software systems grow in size and as versions accumulate, there is a growing body of data (source code) that can be investigated. The growth in this body of data also brings computational challenges such as calculating complexity metrics for each source file in each version of the system, and doing so in a reasonable amount of time; or comparing the similarities across files to identify duplicate software code. As many software systems are developed using a variety of programming languages, there are then issues of aggregating and comparing measures across programs written in different languages.

Software evolves for various reasons. One of the most important is its evolution to accommodate gradually more complex functions. The evolutionary process is endorsed and supported by certain development methodologies, especially agile techniques [1], which rely on a continuous process of rewriting or *refactoring* [2], but also more generally in top-down development methodologies, e.g. [3]. We can identify two distinct approaches to reconstructing this evolutionary process: *syntactic*, where only the raw text is considered, and *semantic*, where run-time behaviours such as sequences of function calls are traced.

In this paper we investigate a syntactic approach, with the application of n-grams to characterise evolution in software source code. An n-gram is a sub-sequence of n items from a given sequence, e.g. a sub-sequence of characters in a word, or a sub-sequence of words in a text. We use the Ferret copy detection technology to measure the similarity between program source files, using n-grams, and this measure gives us an indicator of the changes that have occurred between, for example, two consecutive versions of a file.

The n-gram approach is an attractive method for analysing source code due to the fine-grained comparisons it allows. Where tools like *diff* work at the level of lines of code, n-gram approaches compare sub-sequences of lexical tokens within a line. Previous authors have commented that *diff*-like implementations, when applied naïvely (e.g. to compare all pairs of files in a system), are too computationally demanding for large scale analysis. Ferret operates in near linear time and our comparisons show Ferret to be between one and two orders of magnitude faster than a naïve application of the *diff* tool.

The remainder of this paper is organised as follows: section 2 briefly explains n-grams and reviews their previous application to software evolution; section 3 provides a more detailed description of the method we have used to apply n-grams to characterising software evolution; section 4 explains the analysis we have conducted; sections 5 and 6 report on two software systems we analysed, the source code for the SAC compiler and the Ferret source code itself; section 7 compares Ferret's speed of computation with *diff* and *wc*, and provides further remarks on Ferret's performance; finally, section 8 discusses our results.

2. N-grams and software evolution

N-grams have a wide range of applications. For example, Tomović et al. [4] have used n-grams to classify and cluster genome sequences. McNamee and Mayfield [5] demonstrate that the retrieval accuracy of an n-gram based method for information retrieval of European languages rivals or exceeds methods that are language specific. Rieck and Laskov [6] have developed and evaluated an n-gram based method of detecting network intrusions. Google are reported to use n-gram models in a wide variety of research and development activities, and in 2006 contributed a large n-gram dataset to the Linguistic Data Consortium[1]. Other applications of n-grams include data compression, plagiarism detection, spelling correction, and de-duplication of large datasets.

Despite the wide applicability of n-grams, we have been unable to identify (i.e., through bibliographic searches) any previous work in software evolution that has taken the approach described here. In previous research, the closest application of n-grams to characterising software evolution appears to have been the use of n-grams in *pre-processing* source code prior to subsequent analysis for software evolution (e.g. [7]). There are, however, a number of related avenues of research. For example:

Code clones and duplicated code. Code clones – fragments of code that are syntactically or semantically similar – are often considered to be indicators of poor software quality. N-grams can be very effective at identifying duplicated code, and code clones that are syntactically similar [8]. We are applying n-grams not to identify clones as such, but to measure the degree of similarity or difference between source files as an indicator of change. Phrased another way, code cloning tends to look at copies of code within a version of the system. By contrast, we are looking at changes between versions of a system.

Similarity metrics. Similarity metrics provide, ideally, quantifiable and objective measures of the degree of similarity of the source code of two (or more) software systems, versions of software systems, or sub-parts of software systems. Yamamato et al. [7], for example, have sought to quantitatively measure the similarity of the many versions of the BSD Unix operating system to reveal evolutionary characteristics of that system. For their investigation, they developed a metric, S_{line}, defined as the ratio of shared source code lines to the total source code lines of the software system(s) being evaluated. In some respects, this is similar to our approach in that we use a ratio of shared tokens to total number of tokens. By analysing at the level of lines of code, Yamamoto et al.'s approach is coarser than ours in the kinds of similarity it will identify.

Yamamato et al. recognised the impractical demands that would be placed on computing resources by using a naïve application of *diff* to compare all pairs of files in the large system they analysed. Consequently, they first applied a fast code clone detection algorithm, *CCFinder*, and then applied *diff* to the resulting file pairs where code clones were found. They estimated the worst-case time complexity of their tool to be $O(m^2n^2 \log n)$ for *m* files of *n* lines.

Of particular relevance to our research is their discussion of Broder's work [9], which uses the exact similarity metric we apply in the current analysis. Broder concentrated on detecting similar documents whereas we seek to characterise evolution using a measure of similarity. Yamamoto et al. made the following conclusions about the approach taken by Broder:

> "... choosing token sequences greatly affects the resulting values. Tokens with minor modification would not be detected. Therefore, this is probably an inappropriate approach for computing subjective similarity metric for source code files." ([7], p. 541)

Based on our previous research, we have concluded that a three-token sequence is robust against many minor modifications of sequences of tokens. In addition, the nature of programming languages means that many of the possible minor modifications are illegal as far as the compiler is concerned. Also, as we discuss in the next section and demonstrate later in the paper, our implementation of n-gram analysis is very efficient in terms of demands on computing resources.

3. Characterising software evolution using n-grams

We have previously used n-grams for two applications [10, 11]: to identify copying between students' written essays, and to identify copying between students' program source code. There are two common properties across these applications. First, the documents being compared are sequences of characters that can be aggregated into words or, more generally, into tokens. Second, it was important to know whether a given document has any or all of its content in common with another document.

[1] http://googleresearch.blogspot.com/2006/08/all-our-n-gram-are-belong-to-you.html

When applying n-gram analysis to identify software evolution, the tokens used for source code are the *lexical tokens* defined by the programming language. For example, the following piece of C++ code:

```
for (int i=0, n=MAX; i<=n; ++i)
obj->Incr();
```

would be divided into the following list of tokens, where each token is separated by a space:

```
for ( int i = 0 , n = MAX ; i <= n
; ++ i ) obj -> Incr ( ) ;
```

Notice how symbols and case are preserved, and that special tokens such as '->' are identified. From this list of tokens we would then extract trigrams (trigrams are n-grams of size 3), the first four trigrams from the above example being:

```
for ( int     int i =
( int i       i = 0
```

Having extracted a list of trigrams, we then compute the similarity between each pair of documents. Similarity is computed using a resemblance metric [9] known as the Jaccard coefficient ([12], p. 299): the ratio of common trigrams between the two documents to the total number of trigrams across those two documents. If *A* is the set of trigrams from document 1, and *B* is the set of trigrams from document 2, then:

$$S(1,2) = \frac{|A \cap B|}{|A \cup B|} \qquad \text{Eq.1}$$

The equation produces similarity scores in the range 0 to 1, where 0 indicates that there are no trigrams common to the two documents, and 1 indicates that all and only the trigrams in each document are present in the other document. The measure is of course symmetric, i.e. S(1,2) = S(2,1).

The similarity score should not be interpreted as a percentage score, e.g. S(1,2) = 0.3 does not mean that document 1 contains 30% of the content of document 2. This is because the trigram analysis does not consider the *frequency of occurrence* of a trigram, only whether a trigram occurs at least once. Where two documents have a similarity score of 1, they may have a different frequency of occurrence of the trigrams. In practice, however, two documents with similarity scores of 1 are very likely to be identical copies.

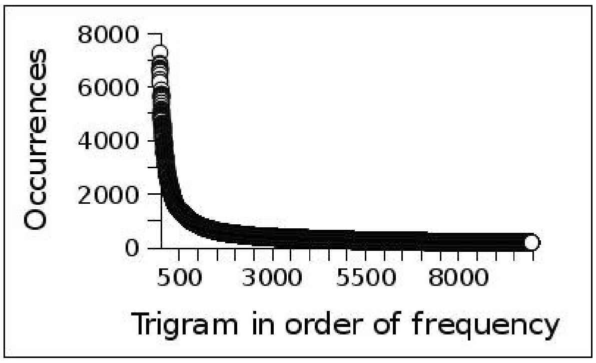

Figure 1 Trigram frequency for the first 10,000 trigrams of the SAC dataset

The importance of trigrams as an indicator of similarity is illustrated by the graph shown in Figure 1. The graph plots the number of times each trigram appears, sorted in order of the most frequent trigram, for the first 10,000 trigrams of the 722,425 trigrams present in the SAC dataset (see section 5 for further details). As is readily apparent, a few trigrams are present in nearly all the files (7819 files make up the dataset), but the number of files in which any trigram occurs rapidly decreases. This exponentially decaying distribution is similar, but not identical, to the Zipfian distribution identified for English words [13] and used in our previous research. The fact that most trigrams occur rarely means that two files with a significant number of trigrams in common are likely to be similar in their syntactic structure, a fact we take advantage of in our analysis of the evolution of source code.

Similarity scores based on n-grams in general, and specifically trigrams in this analysis, can be used to indicate various characteristics of software evolution. For example, at a file level the distribution of similarity scores for a series of versions of a given file can indicate types of change for that file. At a system level, the distribution of similarity scores could indicate the 'growth' occurring over time. Similarity scores directly apply at the file level which is typically the class or module level. In principle, similarity scores can be aggregated for all classes in a package, or for all modules in a sub-folder to provide a higher level of abstraction of software evolution.

Our implementation, Ferret[2], makes it easy to compare large collections of documents for signs of copying. It is also very fast: the algorithm it uses is almost linear in performance, both in memory space required and in time taken, as the total number of words in the input documents grow. Comparisons of the Ferret algorithm with other approaches [14, 15]

[2] Available at: http://homepages.feis.herts.ac.uk/~pdgroup/

show that Ferret's performance is excellent. The reason that Ferret is so fast is that we build an index of trigrams as the documents are read, and then only process the index, so if there are *n* input files, checking all *(n.n-1)/2* pairs of documents is done in almost linear time.

4. Analysis of software evolution

The basis of our analysis is the comparison of files in different versions of an evolving software system. The comparison is performed using Ferret, as described above, and the similarity scores are obtained using Eq. 1. We consider three types of analysis: changes in an individual file over time, patterns of change in each release, and splitting and merging files. The first two types of analysis are to examine the overall history of a software project or individual files; these analyses help to identify trends and typical behaviour, as well as characterise kinds of evolutionary sequences for files or projects. The third type of analysis is a way of examining a particular kind of discontinuity in a file's lifetime, which may arise from a refactoring such as splitting a class into smaller subclasses ([2], p.149). These analyses were performed by processing the output table of similarities from Ferret with Ruby scripts.

4.1. Changes in file over time

We look at each file, and how it changed over time. Where the same named file exists in consecutive versions, we compute the similarity score between these two versions. We look for two effects: the characteristics of the file's lifetime, and whether the file has undergone significant regression, returning to a previous form.

For the analysis of the characteristics of a file's lifetime, we investigate whether the file is stable, revised, or continuously changing. We deduce these states by looking at how the similarity score for that file varies across the versions of that file. For example, a file which remains unchanged will maintain a high average similarity score across its lifetime. Conversely, a file which changes a lot will have a low average similarity score.

Scripts were written to analyse how the similarity scores varied over time. We computed the overall mean and standard deviation of the complete set of similarity scores. We then used these figures to define 'high' and 'low' values for individual files. A 'low' standard deviation is one which is less than the average, and a 'high' standard deviation is more than the average. Our basic categorisation was between those files that are relatively stable over time and those that change frequently. The first group is further separated into those that change very little (i.e. are stable) and those that occasionally change significantly.

Together, Ferret and the scripts highlight three types of file:

1. *Stable files*, defined by a high mean similarity score and a low standard deviation in that score. and *all* changes remain within one standard deviation of the mean.
2. *Revised files*, defined by a high mean similarity score and a low standard deviation and *one or more* changes are more than one standard deviation below the mean.
3. *Changeable files:* If there is a low mean similarity *or* a high standard deviation less than the average mean, then the file has experienced a large amount of change over its lifetime.

We can also explore the rate at which files change from their original state, by considering the similarity of each version of a file with its original form. This gives us a picture of how much files change over time, or their average rate of evolution.

Finally, we check for regression of a file, by which we mean whether a file has returned to its previous form, by looking for a high similarity of a file with a later version where it is dissimilar to previous versions.

4.2. Patterns of change in each release

By plotting the mean similarity measure between versions we can characterise the nature and scope of the changes in each new version e.g., releases where very few changes have occurred compared to releases where almost every file has changed extensively. This can be related to other information collected on the system, such as changes in the number of files for the overall system.

4.3. Splitting/merging of files

For this analysis, we attempt to automatically identify where files have been split into smaller blocks (the treatment of merged files is similar, with a time reversal on the versions). The idea is to set a threshold of similarity based on the overall average similarity, and locate those files which have a higher than average similarity to a potential source document; those identified files are then potentially the results of a split in the file. This technique enables us to automatically identify a popular form of refactoring, where a complex class or file is subdivided into a set of simpler classes or files. Counting the number of such changes may enable us to quantify the way in which complexity is managed by the development team.

4.4. A summary of the systems analysed

We test the Ferret tool on two sets of source code, SAC[3] (Single Assignment C) compiler, and Ferret itself. Table 1 provides simple characteristics of the two sets of source code. We briefly describe the source code and the related executable systems here, and then report our analysis in the next two sections.

Table 1 Systems analysed in this study

System	Versions	Files	LOC	Avg. LOC/file
Ferret	20	190	76K	400
SAC	39	7819	6.7M	860

SAC is a strict purely functional programming language whose design is focused on the needs of numerical applications. It is under development at the University of Hertfordshire, in collaboration with several other Universities world-wide. The C programming language is used as an intermediate language for SAC, in order to achieve portability among different target architectures and to reuse existing compiler technology for the generation of machine specific code. A large compiler project for the compilation of SAC programs into C programs constitutes part of the research being undertaken in the context of SAC. The SAC compiler has been under development since 1995 and has grown from approximately 15 files to approximately 320 files of C code. It is the compiler that is the target of our analysis here. Version control for the compiler is managed using the Subversion (SVN) revision control system.

For the current analyses we 'cut' 39 versions of the compiler, from the approximately 15,000 revisions in Subversion, between the 1st January 1995 and 1st January 2007. Due to the relatively small size of the compiler in the earlier years, we 'cut' only two versions per year from the years 1995 through 1999, and then for the years 2000 through 2007 we cut four versions per year.

Ferret has grown from an initial set of 6 files to 26 files. The initial version of Ferret analysed here was already release 2.0, the earlier history is no longer available. Due to space restrictions for this paper, we only summarise our results for Ferret in section 6.

[3] For more information visit: http://www.sac-home.org/

5. Analysis of the SAC source files

5.1. Changes in file over time

There were 435 different files which appeared in subsequent releases of the SAC system. The average similarity across all the version changes for the complete system was 0.93 with a standard deviation of 0.098. This suggests that most of the source code remains constant between releases, at least at the granularity at which the SAC versions were cut.

Of the 435 files, 84 were considered stable, 155 revised, and 196 changeable. Figure 2 shows a sample of change history over time for each of the three types of change defined in section 4.1. The file `LoopInvariantRemoval.c` shows the effects of revisions in initial versions, interspersed by a few versions of constancy. This observation is supported by a developer's comment that the compiler optimization implemented in this file has undergone some redesigns in order to improve its applicability. The file `Modulemanager.c` has been rewritten several times as its functionality needed to be extended substantially whenever language extensions were made. This file is a good example of the changeable type of file. The `Boundcheck.c` file is found to be a stable file. In fact, it served the code generation for out-of-bound checks which constitute a rather straight-forward piece of code.

We also analysed the degree of evolution of a file by comparing the similarity of the same named files across larger numbers of versions. For instance, a file which went through five versions would have its similarity compared between versions 1 and 2, versions 1 and 3, versions 2 and 4, etc.

Figure 3 shows the average similarity, and error bars to one standard deviation, of the initial 13 files across the entire lifetime of SAC (1 file was removed after 4 versions, but 8 files were retained throughout the complete lifecycle). The figure clearly shows how files rapidly evolve, retaining few, if any, features in common with their initial release.

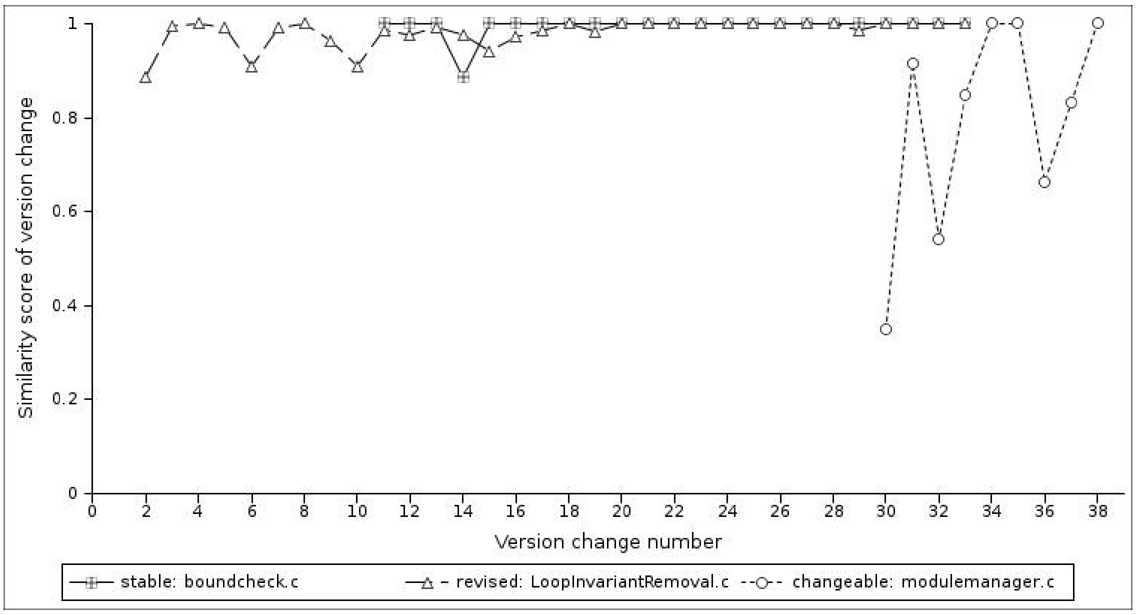

Figure 2 Examples of the three types of change, for the SAC dataset

1018 out of the total set of 7819 files showed signs of ‘regression’, by which we mean that a later file is more similar to an earlier file than an intermediate version. However, although easy to check, the kind of changes in these files was rather small. We did not detect any major reversions of a file back to an earlier version.

5.2. Patterns of change in each release

Figure 4 presents the average similarity between releases of the same file for all files across all versions. The clear drop in average similarity at versions 7, 30 and 34 suggests that substantial changes occurred in these releases. Notice the larger standard deviations also for releases 7, 30 and 34. There was an almost constant period between releases 22 and 28 as indicated by the constant average and the low standard deviations. Notice also the sharp climb in average similarity for versions 1 through 4 as the system was being initially developed.

Figure 4 can be compared with the accumulated number of .c files for the SAC compiler, presented in Figure 5. In Figure 5, the ‘bump’ from versions 29 to 34 occurs at the same time as the increased deviations in Figure 4 for that period. Figure 5 provides a measure of external change to the number of files in the system, and Figure 4 provides a measure of internal changes to a file. Therefore these are quite different measures that complement each other, and are consistent in that both metrics suggest increased code change activity. The dates associated with the ‘bump’ correlate precisely with the major refactorings that have been executed during the lifetime of the project so far. The relative plateau in the accumulated number of files for versions 22 through 29 in Figure 5 are, again, consistent with the stable mean and low deviations in Figure 4.

5.3 Splitting files

The splitting and merging of files has already received considerable attention. For example, Godfrey et al. (e.g. [16]; [17]) have reported a number of investigations based on their concept of origin analysis, and Antoniol et al. [18] have investigated class evolution discontinuities using an approach inspired by vector space information retrieval. We acknowledge[4] that our investigation of file splitting and merging reported here is therefore immature. The purpose of investigating the merging and splitting of files is to explore the limits of Ferret’s usefulness to investigating software evolution.

A script was developed to locate candidate files which had been split. The motivation here was to look for files which had grown in complexity and then had been subdivided into smaller blocks.

[4] We also thank the reviewers for their comments here.

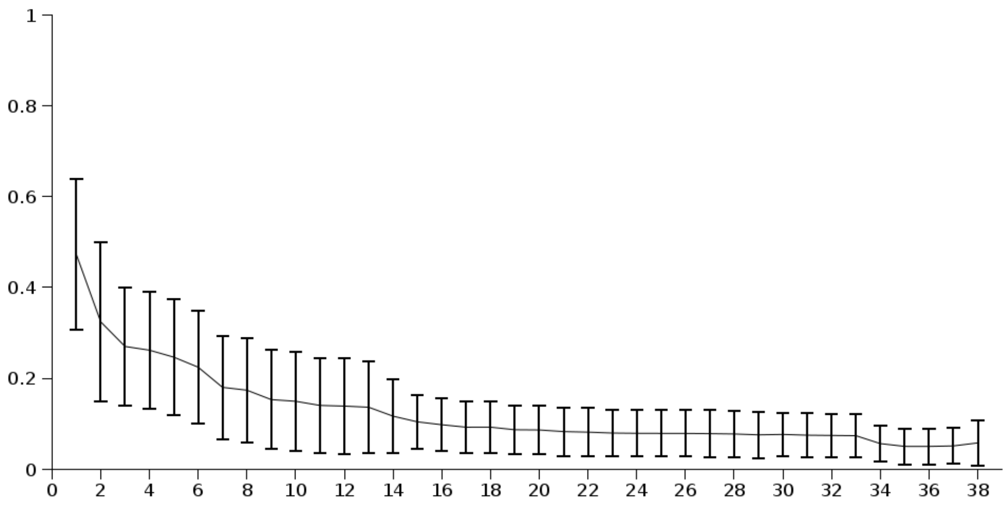

Figure 3 Similarity of subsequent versions of a file to its initial release

We first extracted all comparisons between files of different names, *but consecutive versions*, to produce a set of 177850 comparisons. The mean similarity between these files was 0.035, with a standard deviation of 0.031. We then extracted all comparisons between files of different names in consecutive versions with a similarity greater than 0.066 (the mean + one standard deviation) yielding 411 candidate split files from the 7819.

Figure 6 shows a sample of the code reused in a different file. The underlined code is the duplicate in the reused file. Notice that the n-gram comparison has been sensitive to small changes in the code (e.g. the names of functions), but identified a sizable block which has been reproduced in the candidate child file.

6. Analysis of the Ferret source files

There were 42 different files which appeared in subsequent releases. The average similarity across all the version changes for the complete system was 0.91 with a standard deviation of 0.16. Of the 42 files, 22 were considered stable, 5 revised, and 15 changeable.

There were 22 files identified as candidate split files. Some of these were clearly caused by renaming of files, which occurred at version 4.0 of Ferret, and are identified by high levels of similarity, around 0.8/0.9. At version 4.4, several files had become large, triggering Fowler's code smell of a 'large class'. Conscious use of the 'Extract class' refactoring practice resulted in several smaller files, present in version 4.5. What is interesting here is that our n-gram approach identifies these files as they have a similarity around 0.2 – 0.4, much higher than the average similarity of 0.034, but clearly not representing exact copies of the files.

7. Evaluation of Ferret performance

Table 2 provides simple metrics on the performance of the Ferret copy detection technology. The duration includes reading in all of the source code, similarity calculations, and writing out the results to file (e.g. for SAC, the output file is 2.7GB). For both systems, the Ferret analysis was performed using a 2.8GHz PC with 1.25GB RAM running Linux. The data structure for the analysis is entirely held in RAM.

Table 2 Performance of Ferret

System	Duration (seconds)	Files	Document pairs	Pairs per second
Ferret	10	190	17955	1795
SAC	2160	7819	30564471	14150

We used *diff* to compare all pairs of files for all versions of Ferret and, separately, of SAC. The *diff* analysis of the Ferret source code files takes 7 min. 18 sec. so Ferret is approximately 43 times faster than *diff*. We also ran the *wc* (word count) program against the same set of files. *wc* is slower than *diff* and this corroborates our conjecture that the major cost in performing the comparisons is simply the time that it takes to process all the files. Because Ferret only processes each file once, it is much faster. We estimated that the *diff* analysis of SAC source code takes approximately 50 times longer than the *diff* analysis of the Ferret source code. In terms of lines of code processed per second, the estimated performance of the Ferret, *diff* and *wc* analysis of the Ferret source code is, respectively: 7600 LOC/sec., 16397 LOC/sec. and 15025/sec. Based on these estimates, Ferret takes twice as long as *diff* and *wc* to initially process each line of code.

Formally, the *diff* tool requires $O(n^2 \log n)$ time to process one pair of files ([19] cited in [7]) where n is the length of the input. By contrast, Ferret requires an estimated $O(n)$ time.

As a further contrast, Neamtiu et al. ([20]) developed a tool to quickly compare the source code of different versions of a C program. Their tool analyses the Abstract Syntax Trees (ASTs) of the source code, these ASTs being generated by the use of CIL [21]. Running on a PC approximately twice as fast as ours, their tool processed 400K LOC in about 70 seconds.

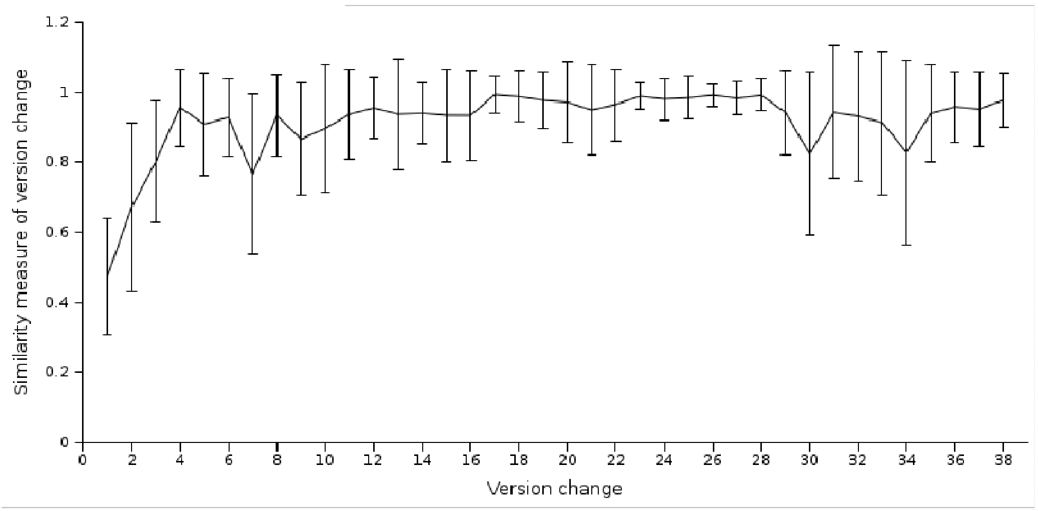

Figure 4 Average similarity between consecutive versions of the same file

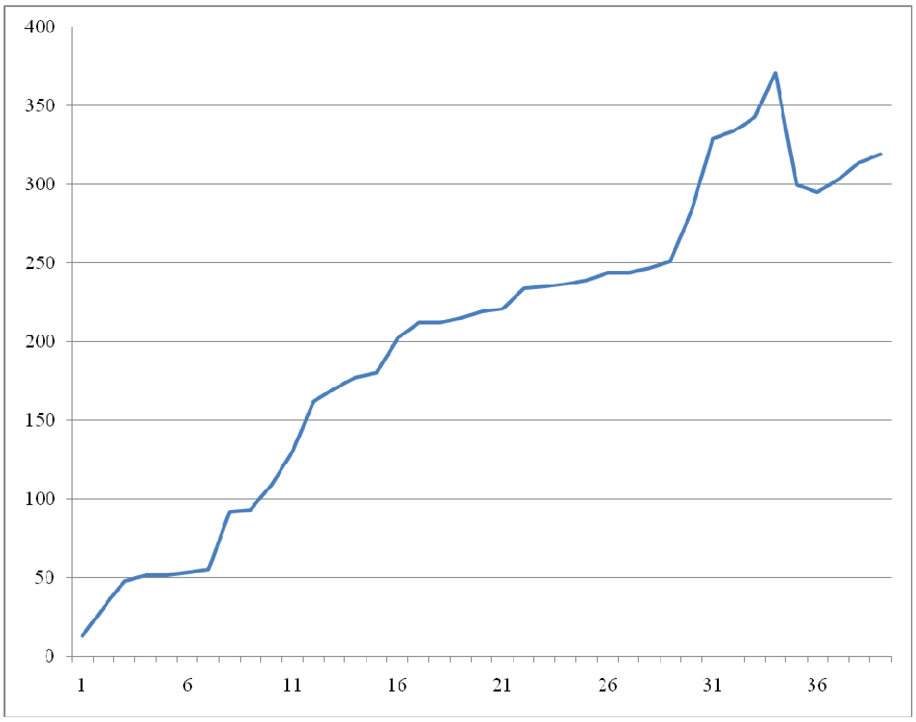

Figure 5 Accumulated number of files per version

```
node *BlocksCons(node *arg_node, node *arg_info)
{
  statustype old_attrib;
  funtab *old_tab;
  DBUG_ENTER( "BlocksCons");
  DBUG_PRINT( "BLKCO", ("begin"));
  DBUG_ASSERT( (NODE_TYPE( arg_node) == N_fundef),
               ("wrong type of arg_node"));
  if ((FUNDEF_BODY( arg_node) != NULL) &&
      (FUNDEF_STATUS( arg_node) != ST_foldfun) &&
      (FUNDEF_ATTRIB( arg_node) != ST_call_rep)) {
    old_tab = act_tab;
    act_tab = blkco_tab;
    old_attrib = INFO_BLKCO_CURRENTATTRIB( arg_info);
}
```

Figure 6 Fragment of reused C code (the underlined code has not been reused)

For both Ferret and Neamtiu et al.'s tool, and in contrast to *diff* and *wc*, the high level of performance is due to the fact that representations of the source code are analysed and not the source code itself; and that the input source files are not needed once the representations have been generated.

8. Discussion

Our main findings are to demonstrate that the n-gram approach is fast and produces meaningful results. Although some of our other findings may not be 'new' to the software evolution community, the fact that our approach is able to generate these standard kinds of metrics in evolution confirms that our approach produces meaningful results when analysing software evolution. Further work may be able to establish additional metrics that can be generated by Ferret.

In terms of our more specific results, we found that:

- For both systems studied, after their initial releases there was subsequently a small amount of incremental growth per release.
- For both systems, there was a high average similarity in subsequent versions of the same file
- Our n-gram analysis is able to differentiate between different types of 'file histories' e.g. stable, revised and changeable. There is however a wide range in the number of files in each type e.g. SAC had 19% of stable files whilst Ferret had 52% stable, and 35% vs. 12% for revised files. Possibly this reflects the fact that SAC is a larger system that has been worked on extensively by a team, whereas the Ferret source code has been under the custody of a single programmer.
- Our n-gram analysis is capable of identifying regressed files and 'reborn' files' i.e. where files 'reappear' in later releases after having been removed from earlier releases. For the two systems we studied, we were not able find examples of the regression and 'rebirth' but we believe this is a characteristic of the systems studied rather than the n-gram approach we used.
- Our n-gram analysis detected cases of split files.
- The Ferret implementation of n-grams is extremely efficient e.g. for the larger system, SAC, this was effectively 14150 pairs of files compared per second.

One of the advantages of the n-gram approach is that it provides a language-independent analysis, and one of the advantages of the implementation of trigram analysis in Ferret is that the analysis is very quick. Although not discussed in this paper, Ferret also provides a graphical user interface to allow the user to examine copying in the source code for any two files of interest. This functionality can easily be modified to output detailed comparisons between files. We are also conscious however that we have reported our analysis for only two relatively small systems.

There are a number of directions in which we can extend this research. We are interested in applying n-gram analysis to other programming languages, and to other *types* of programming language, such as functional languages. We also want to apply our n-gram approach to larger software systems, such as the various BSD operating systems investigated by Yamamoto et al. [7]. Our research on the application of n-gram analysis to plagiarism detection has indicated that three-token sequences (trigrams) are sufficiently effective, but we note that McNamee and Mayfield [5] used 4-token sequences for their investigation of information retrieval, and Broder [9] used larger sequences again, so there may be benefits for characterising software evolution using longer sequences. Finally, we want to compare the performance and effectiveness of Ferret against other syntactic approaches, but also to consider how Ferret can complement semantic approaches.

9. Conclusion

We have demonstrated the application of n-grams to characterise the evolution of software code by applying the Ferret copy detection tool, which computes similarity based on trigrams, to two software systems, SAC and Ferret. Our trigram analysis has been able to differentiate different types of changes to source files, characterise the history of changes to individual files, identify file splitting and, in principle, identify regressed and 'reborn' files. The Ferret implementation of n-grams is extremely efficient, operating at one to two orders of magnitude faster than *diff*. For the larger system SAC, this was effectively 14150 pairs of files compared per second. We believe our results are sufficiently encouraging to warrant further research and tool development in the study of software evolution.

Acknowledgements

We thank the three reviewers for their constructive criticisms of an earlier version of this paper. SAC not yet open source and therefore is only available on individual request by contacting the SAC project team (info@sac-home.org).

References

1. Beck, K., *Test-driven development: By example*. 2003, Reading, MA.: Addison-Wesley.
2. Fowler, M., et al., *Refactoring: Improving the Design of Existing Code.* 1999: Addison-Wesley Professional.
3. Milewski, B., *C++ in action: industrial-strength programming techniques*. 2001, Upper Saddle River, NJ.: Addison Wesley.
4. Tomović, A., P. Janičić, and V. Kešelj, "n-Gram-based classification and unsupervised hierarchical clustering of genome sequences". *Computer Methods and Programs in Biomedicine*, 2006. 81(2): p. 137 - 153.
5. McNamee, P. and J. Mayfield, "Character N-Gram Tokenization for European Language Text Retrieval ". *Information Retrieval*, 2004. 7(1-2): p. 73-97.
6. Rieck, K. and P. Laskov, "Detecting Unknown Network Attacks Using Language Models". In *Detection of Intrusions and Malware & Vulnerability Assessment*. 2006, Springer Berlin / Heidelberg. p. 74-90.
7. Yamamoto, T., et al., "Measuring Similarity of Large Software Systems Based on Source Code Correspondence". In *Product Focused Software Process Improvement (PROFES 2005)*, F. Bomarius and S. Komi-Sirviö, Editors. 2005, Springer Berlin / Heidelberg.
8. Bellon, S., et al., "Comparison and Evaluation of Clone Detection Tools". *IEEE Transactions on Software Engineering*, 2007. 33(9): p. 577 - 591.
9. Broder, A.Z., "On the resemblance and containment of documents". *Proceedings of Compression and Complexity of Sequences*, 1998: p. 21-29.
10. Bao, J.P., C.M. Lyon, and P.C.R. Lane, "Copy detection in Chinese documents using Ferret". *Language resources and evaluation*, 2006. 40: p. 357-365.
11. Lyon, C.M., R. Barrett, and J.A. Malcolm. "A theoretical basis to the automated detection of copying between texts, and its practical implementation in the Ferret plagiarism and collusion detector". In *JISC conference on plagiarism: prevention, practice and policies*. 2004.
12. Manning, C.D. and H. Schütze, *Foundations of statistical natural language processing*. 2001, Cambridge, MA.: The MIT Press.
13. Zipf, G.K., *Human behavior and the principle of least effort*. 1949, Cambridge, Mass: Addison-Wesley Press
14. Bao, J., et al., "Comparing different methods to detect text similarity". 2007, Science and Technology Research Institute, University of Hertfordshire, Technical Report CS-TR-461.
15. Lyon, C., R. Barrett, and J. Malcolm, "Experiments in electronic plagiarism detection". 2003, Computer Science Department, University of Hertfordshire, Technical Report CS-TR-388
16. Godfrey, M.W. and L. Zou:, "Using Origin Analysis to Detect Merging and Splitting of Source Code Entities" *IEEE Transactions on Software Engineering*, 2005. 31(2): p. 166-181.
17. Godfrey, M.W. and Q. Tu. "Tracking Structural Evolution Using Origin Analysis". In *International Workshop Principles of Software Evolution (IWPSE-02), May 2002*. 2002.
18. Antoniol, G., M.D. Penta, and E. Merlo. "An Automatic Approach to identify Class Evolution Discontinuities". In *7th International Workshop on Principles of Software Evolution (IWPSE'04)*. 2004.
19. Hunt, J.W. and M.D. McIlroy, "An algorithm for differential file comparison". 1976, Computing Science, Bell Laboratories, Murray Hill, New Jersey.
20. Neamtiu, I., J.S. Foster, and M. Hicks. "Understanding Source Code Evolution Using Abstract Syntax Tree Matching". In *International workshop on mining software repositories (MSR '05), May 17, 2005*. 2005. Saint Louis, Missouri, USA.
21. Necula, G.C., et al., "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs" In *11th International Conference on Compiler Construction (Lecture Notes in Computer Science 2304)*. 2002, London, UK Springer-Verlag p. 213-228.