CVSSearch: Searching through Source Code using CVS Comments

Annie Chen, Eric Chou, Joshua Wong, Andrew Y. Yao, Qing Zhang, Shao Zhang, Amir Michail
School of Computer Science and Engineering
University of New South Wales
{anniec,tzuchunc,joshuaw,andrewy,qzha132,shaoz,amichail}@cse.unsw.edu.au

Abstract

CVSSearch is a tool that searches for fragments of source code by using CVS comments. CVS is a version control system that is widely used in the open source community [10]. Our search tool takes advantage of the fact that a CVS comment typically describes the lines of code involved in the commit and this description will typically hold for many future versions. In other words, CVSSearch allows one to better search the most recent version of the code by looking at previous versions to better understand the current version. In this paper, we describe our algorithm for mapping CVS comments to the corresponding source code, present a search tool based on this technique, and discuss preliminary feedback.

1 Introduction

Search tools for source code are important in software maintenance activities [22]. However, if the code is poorly commented, then using a standard search tool, such as grep [16], is problematic. For example, it may be obvious from using the application that it has cut and paste functionality, though it may not be at all obvious how to use grep to find lines that implement this functionality. Since we can no longer depend on matching words in comments, we must use the search tool to match the code itself — which is difficult if we have never seen the code before. Indeed, within the context of component retrieval, Maarek et al. write:

> An examination of numerous samples of code allowed us to reach the conclusion that some useful information can be extracted from programs written in a high-level language using good programming style, whereas little conceptual information can be found in typical real-world code chosen at random. Unfortunately, even when dealing with well-written code, there is a very low probability that the programming styles of the various pieces of code will be consistent. Even a single programmer may use totally different identifiers for expressing the same concept from one day to another... Another limitation comes from the fact that there are many more possibilities for identifiers than for natural-language words, since they do not follow any morphological or syntactic rules. [19, p. 801–802]

Maarek et al. conclude: "In other words, when there is no way to guarantee good, let alone consistent and compatible, programming styles, extracting attributes from raw code does not give significant results. Therefore we prefer concentrating on the other possible source of information; i.e., the natural-language documentation either inserted into the code — the comments — or associated with the code, e.g., manual pages [19, p. 802]."

If the code is well commented, one might expect standard search tools to work well. After all, comments are intended to not only state the purpose of the various pieces of code, but also to explain how that code works. In other words, they provide information at various levels of abstraction. Returning to our example, one would expect that doing a grep on "cut|paste" would match these words in comments and thus return those sections of code responsible for implementing cut and paste functionality.

However, naively searching through comments is problematic for various reasons. As Maarek et al. note, relating comments to the portion of code they concern is a very difficult task in free-style code. They write:

> Indeed, in free-style programming, programmers can insert comments wherever and in any format and any length they wish. Although comments usually describe the containing routine or the one just below, in general it is impossible to automatically determine what part of the code is covered. [19, p. 802]

Because of this problem, Maarek et al. resort to using external documents associated with code — specifically, manual pages — to construct profiles for reusable components.

Their work assumes that it is easy to determine which component a manual page refers to — as is the case with Unix-like manual pages for example.

Antoniol et al. [2, 3] address cases where this assumption does not hold: namely, it may not be easy to know which code an external document refers to. Consequently, they have built a system that recovers code to documentation links in object-oriented systems. Indeed, they note: "traceability links between the requirement specification document and the code are a key to locate the areas of code that contribute to implementing specific user functionality [3, p. 137]."

In this paper, we present a general purpose search tool, CVSSearch, that also leverages natural language documentation — namely, CVS comments. As far as we know, this is the only tool of its kind.

CVS is an open source version control system that is widely used in the open source community [10]. Indeed, almost any large open source project makes use of CVS, especially if multiple developers are involved.

CVS comments provide a good source of documentation. While open source code may not always be well-commented, large open source applications almost always have very good CVS comments, particularly when many developers are involved.

Our CVS-based search tool takes advantage of the fact that: (1) a CVS comment typically describes the lines of code involved in the commit; and (2) that this description will usually hold for many future versions. In other words, CVSSearch allows one to better search the most recent version of the code by looking at previous versions to better understand the current version.

Elaborating on point (1), observe that a comment in a CVS commit not only describes the change made but also indirectly describes the purpose of the lines of code involved in that change (e.g., "added footnote feature" indirectly reveals that the lines involved in the commit have something to do with footnotes).

With respect to point (2), we note that the purpose of lines usually does not change often — even if the contents of the lines do. For example, mouse handling code will remain just that in many future versions even if some details change throughout the evolution of an application.

Our approach solves several of the problems discussed with finding useful functional information for code:

- by searching through natural language instead of (possibly uncommented) code, we avoid the problems of trying to extract useful functional information from free-form code with ad hoc naming conventions; and
- by using CVS comments, we automatically have an explicit mapping of the commit comment and the lines of code that it refers to.

Yet, CVS comments have additional benefits. A fragment of code may be involved in multiple commits, in which case, it would have several commit comments associated with it. In such a case, we can view each commit comment as providing yet another summary/aspect of that fragment (or parts thereof). Consequently, it is possible to address the vocabulary mismatch problem — a query word will match a line if at least one developer thought of using that word to describe the changes made. In contrast, a comment in the code provides just one summary of the code it describes — so there is less opportunity to match the query word.

We have observed that developers are more likely to write CVS comments than actual code comments. We believe this to be the result of two factors. First, open source developers often use CVS comments as an opportunity to describe their changes to other developers so that everyone is made aware of progress and development directions. Second, CVS comments need not be of as high quality as code comments since few people will see them, so open source developers are more likely to write them quickly without worrying too much about whether they are of sufficient quality to avoid embarrassment.

Finally, CVS comments capture information that is typically not found in code comments. For example, if one fixes a bug, one does not typically write a code comment saying that a bug has been fixed. Yet, CVS comments typically do contain such information. So, one can use a search tool based on CVS comments to search for code that is bug-prone, as those lines would have been involved in many commits with a "bug fixed" commit comment say. Perhaps more interestingly, CVS comments provide motivation and history for why that code is the way it is. While such information may not necessarily help users find code fragments, it will help them understand the code that they do find using CVSSearch.

The remainder of the paper is organized as follows. Section 2 explains how we associate CVS comments with lines in the most recent version of the code. Section 3 describes our tool, CVSSearch, which we have used on real-life KDE applications of significant size. Section 4 presents preliminary evaluation of our tool. Section 5 discusses related work. Section 6 summarizes the paper, concluding with future work.

2 Technique

To search for lines of code by their CVS comments, we produce a mapping between the comments and the lines of code to which they refer. Here we are only interested in the lines of code found in the newest version of each file. Unlike the `cvs annotate` command, which shows only the last revision of modification for each line, we record all revisions of modification for each line. For example,

if line 3 in the newest version of the file first appears in revision 1.2, and is subsequently changed in revisions 1.4 and 1.5, then we need to associate line 3 with the comments of revisions 1.2, 1.4 and 1.5.

2.1 Algorithm Overview

Consider a file f at version i which is then modified and committed into the CVS repository yielding version $i + 1$. Moreover, suppose the user entered a comment C which is associated with the triple $(f, i, i + 1)$.

By performing a diff between versions i and $i + 1$ of f, we can determine lines that are modified or inserted in version $i + 1$; we associate comment C with all such lines. (Figure 1 shows such a diff, visually, between two successive versions of a file where $i = 48$; modified or inserted lines in version 49 are shaded.)

However, given we are interested in searching the most recent version of each file, we need a *propagation phase* during which the comments associated with version $i+1$ of f are "propagated" to the corresponding lines in the most recent version of f, say $j \geq i + 1$. This is done by performing diffs on successive versions of f to track the movement of these lines across versions (even in the presence of changes to the lines themselves) until we reach version j. (Figure 1 shows the final outcome of this propagation phase in the third file which has version $j = 68$. Observe how the lines are matched up across versions 49 and 68 even though the line numbers have changed due to deletions/additions of preceding lines in the file over time.)

2.2 CVS Comments

The following CVS commands and their outputs are used to produce the required mapping between CVS comments and code.

- the `cvs log` command displays log information for files:

```
RCS file: /repository/file.h,v
Working file: file.h
head: 1.5
...
description:
----------------------------
revision 1.5
date: ...
cvs comment ...
----------------------------
revision 1.4
date: ...
another cvs comment ...
----------------------------
...
```

- the `cvs diff` command shows the differences between files in the working directory and the source repository, or between two revisions in the source repository:

```
...
RCS file: /repository/file.h,v
...
9c9,10
< old line
---
> new line
> another new line
```

We first associate the CVS comments with the corresponding revision by parsing the `cvs log` output to get $\{(revision, cvs\ comment)\}$. We then use the diff output to associate the cvs comment for each revision with the lines involved in that commit, as discussed in the previous section. The output of the diff command is also used in the propagation phase.

2.3 Propagation Phase

The mapping algorithm for the propagation of comments may start from the earliest revision and go forward, or start from the latest revision and go backward. Currently we are only interested in the main branch of the CVS repository. In this paper, we will present the "backwards" algorithm, which avoids unnecessary profiling of deleted lines, empty lines, removed files, etc.

We shall step through this algorithm with a simple example file, `bob`, which has three revision, v1.3, v1.2, and v1.1. The content of `bob` looks like this:

Line \ Revision	1.3	1.2	1.1
1	a	a	a
2	b	b	b
3	c	c	-
4	+	d	-
5	+		d
6	d		

1. `cvs log -b bob` shows that the file has revisions 1.1, 1.2, and 1.3 with their corresponding comments: $\{(revision, cvs\ comment)\}$.

2. compare all consecutive pairs of revisions using the `cvs diff`[1] command, ie. 1.3 and 1.2 as well as 1.2 and 1.1.
   `cvs diff` produces three types of output: addition, change or deletion. e.g suppose `cvs diff -r1.2 -r1.3 bob` produced the following output:
   ```
   ...
   3a4,5
   ...
   ```

[1] We will be using a variation of diff to produce a more accurate mapping, as will be described shortly.

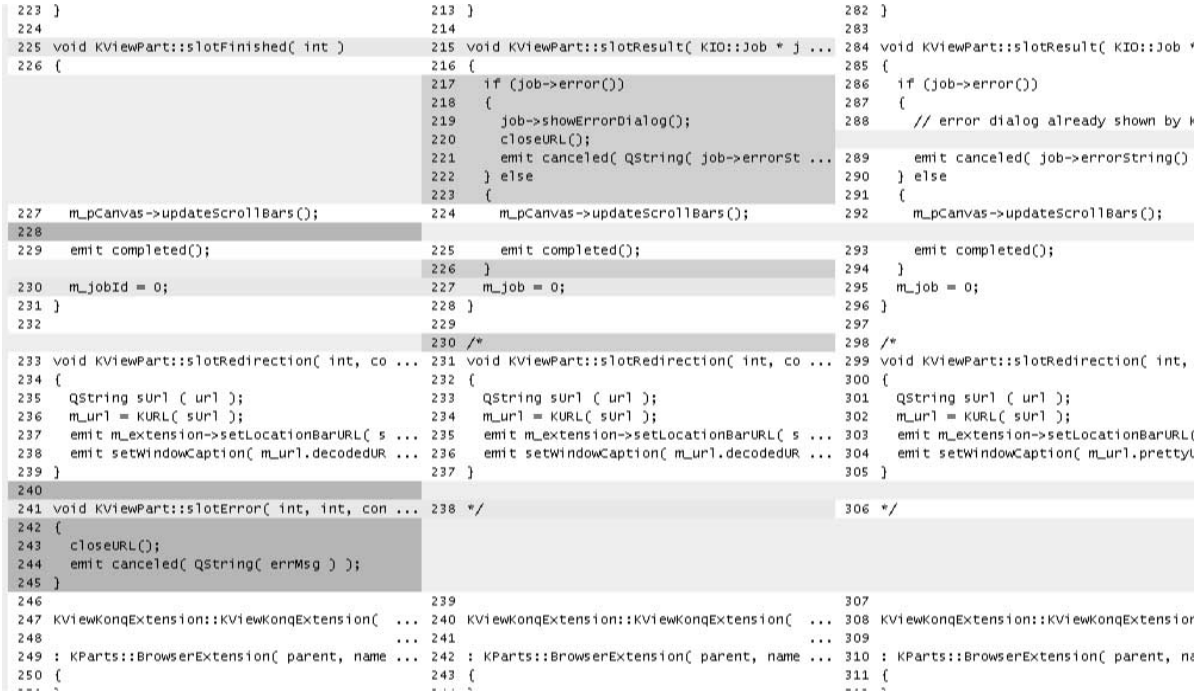


Figure 1. Three versions of kview_view.cc are shown: (1) v. 48 before commit on left; (2) v. 49 after commit in middle; and (3) v. 68 (most recent) on right.

this shows that lines 4 to 5 is added in revision 1.3, so they should be mapped to revision 1.3 (and be associated with that revision's comment). However, as explained earlier, line numbers can change across revisions because of insertions and deletions, so we need to track the movement of each line. That is, we work out which line in the latest version corresponds to which line in each revision, so the mapping is $\{(cur\,line, latest\,line)\}$.

3. combine the results from 1 and 2 to get $\{(latest\,line, cvs\,comment)\}$ for file `bob`.

More on Step 2 - The Backward Mapping algorithm

We traverse every version of a given file backwards (i.e. from the most recent to the least recent) and keep track of lines across versions by maintaining an array that has an element for each line in the version currently being examined. Each element is either a positive integer representing the corresponding line in the latest version, or -1 if the line no longer exists.

Initialization of the mapping array

Since we are going from the latest version, the initial mapping array is initialized as its own index. So for our example, the array is initialized as below:

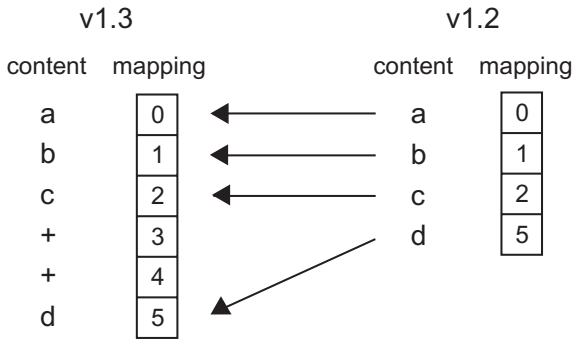
v1.3

content	mapping
a	0
b	1
c	2
+	3
+	4
d	5

Marking of unwanted lines, for example, empty lines, can also be done in the initialization step by setting the value of the relevant elements to -1.

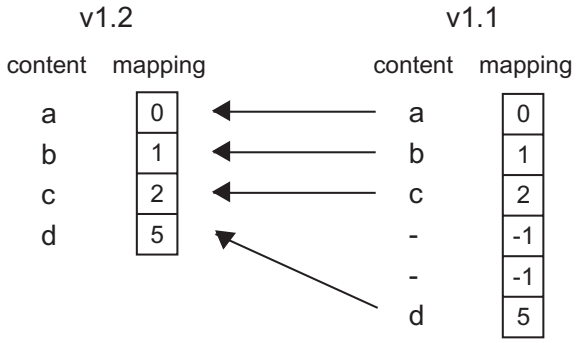
Addition of lines

Using `cvs diff` as described before, we find that between v1.3 and v1.2 two lines have been added after line 3, so we know that from lines 1 to 3 the mapping will be the same as v1.3, but after line 3 we need to skip two lines.



Deletion of lines

From v1.2 to v1.1 two lines have been deleted. We map lines before line 3 to the same index as usual, and we then mark two indices after line 3 to -1. Later when we are storing comments, -1 tells us that this line has been deleted in newer versions, hence we do not need to store comments for this line.



Changed lines

When a line changes, the mapping of this line remains unaffected; this is where multiple comments across different revisions are accumulated. For example, a section of code is inserted in v1.2 for creating menu bar. Later in v1.4, the same block of code is modified to fix a display bug, hence that block of code should have both "menu bar" and "display bug" associated with it. However, a problem occurs when changes span multiple lines. For example:

Line	Older Version	Newer Version
1	a	a
2	b	b
3	cat	added a line
4	d	sleepy cat
5		added a line
6		d

Using `cvs diff` would give us:

```
3c3,5
< cat
---
> added a line
> sleepy cat
> added a line
```

This only shows that line 3 in the older version is changed to lines 3-5 in the newer version, but we can see from the file that "cat" should be matched to "sleepy cats", so more ideally, the output should tell us that one line got added after 2, line 3 got changed, and one line got added after 3, i.e.

```
2a3
> added a line
3c4
< cat
---
> sleepy cat
3a5
> added a line
```

We solved this problem by using a modified version of diff, which applies our own string alignment algorithm (see Appendix) to changed blocks of code to achieve a more accurate mapping.

2.4 Database Storage and Querying

We have chosen the MG (Managing Gigabytes) [25] system for our database because it provides fast text retrieval on large text based database. It also provides stemming and ranking for query results using cosine similarity [25, p. 185].

For each line of a file we store into MG its associated CVS comments. When given query words, MG returns all lines whose CVS comments contain at least one of those words. Moreover, the results are ranked so that lines returned first tend to contain the most query words — and with multiple occurrences of them. In fact, MG returns a score along with each line match to indicate the quality of the match with respect to the query.

When user queries the database, we combine the results given by MG with matches from applying grep on the source code. For example, if the user searches for the words "cut paste", then we also perform a grep query "cut|paste" to look for lines of code that contain one of those words. We have found that such grep matches tend to complement CVS comment matches: one looks at code while the other looks at what people say about that code.

Our tool actually returns a ranked list of files first, where the score of a file is computed as $\sum_{i>j} \frac{S_i * S_j}{(i-j)^2}$, where the summation is over all lines that matched the query from that file and where S_k indicates the score of line k. (The score for a grep match is half the maximum score returned by MG

for that file.) This formula rewards those files that tend to have many matches close together where the matches themselves have high scores. A user can then click on a file to see the matched lines, which are color coded to indicate the strength of each match.

3 The Tool

We have built a web-based demonstration tool, CVSSearch, that allows users to search through 30 KDE projects.[2] Users can enter keywords to search for and select a project to search in. Ranked results are then displayed, along with the number of lines matched in each file; one count for grep and another for lines matched in cvs comment. (See Figure 2, (a) where the user has searched for "password" in the kmail application.)

A user can then click on a file to examine its line matches. (See Figure 2, (b) where the user has clicked on the first file in (a).) Matches are shown on the left with a tag displaying the type of match, i.e. CVS, grep or both. Moreover the matched lines are highlighted according to how strong the match is. Darker highlight denotes stronger matches and vice versa. (Observe that many of the CVS matches are relevant to the "password" query even though the word "password" doesn't appear in those lines.)

On the right, the tool shows the source code for that file. A user can click on any of the matched lines on the left to bring the source code on the right to that particular line so the user can examine that line in its context. In the bottom frame the associated CVS comments of the selected lines are displayed, so user can see what was matched in the comments for that line. (In Figure 2, (b), the user has clicked on line 318, a CVS match, with the associated CVS comment shown in the bottom frame.)

We conclude this section with some statistics for five KDE applications analyzed by our tool. (See Figure 3.) For each application, we show its size in terms of lines of code, the average number of revisions per file, the time it took to analyze the application and build the CVS comment database, the size of that database, and the average number of CVS comments associated with each line in the most recent version. Observe that we have tried CVSSearch on real-life applications of significant size and significant evolutionary history.

4 Preliminary Evaluation

We conducted a preliminary evaluation of our tool by 74 students at the University of New South Wales [7]. The students were asked to perform at least 2 queries for each of the following KDE applications: konqueror (a web browser), kmail (an email client), knode (a newsreader), korganizer (a scheduling program), and kword (a word processor). (See Figure 3 for applications statistics.) Moreover, they were instructed to use reasonable queries given the application type. For each query submitted, students were to indicate whether CVSSearch alone or grep returned better results — or whether they were both about the same. This should be clear since CVSSearch informs the user whether line matches were obtained from CVS, grep, or both. (See Figure 2.)

The students were not familiar with the source code of these KDE applications so our survey is testing how well CVSSearch compares with grep on unfamiliar code. In future surveys, we shall consider both familiar and unfamiliar code.

Table 1 shows the results of the query evaluations, split by application type, as well by total. Of the 703 queries submitted across all applications by students, 40% were better handled through CVS comments, 32% were better handled by grep, and 28% were handled equally well by both approaches.

We also calculated p-values for both "CVSSearch being better" and "grep being better". We see that the results from the 703 evaluations indicate that CVSSearch is actually more likely than grep to give better answers overall, with a p-value of 0.0143. This is strong indication that overall, CVSSearch performs better searches than grep using CVS comments alone (e.g., without looking at the code itself using grep). Moreover, if we look at the breakdown by application type, we see that there is strong indication that CVSSearch outperforms grep for kmail and kword with p-values of 0.0249 and 0.000497, respectively. Moreover, there is some weaker evidence that CVSSearch is also better with konqueror and knode, with p-values of 0.104 and 0.400, respectively. However, CVSSearch clearly loses to grep on korganizer, where grep has a p-value of 0.0141. So, while all queries combined indicate that CVSSearch is better than grep overall, the results are less clear for particular applications. We suspect that a combination of factors lead to this variation: the quality of CVS comments used, the quality of identifier names, as well as the student's familiarity with the application type.

Student remarks were particularly interesting. One student wrote "It searches a variety of related and relevant words, for example, when 'date' is searched, results including 'time' are also returned." This property follows naturally from our approach since many lines with words such as "time" also have associated CVS comments with "date". Another person wrote "It (CVSSearch) helps locate parts of the code better than grep, especially on conceptual ideas, where they (are) most likely mentioned in the CVS comments."

[2] KDE is a powerful Open Source graphical desktop environment for Unix workstations. See http://www.kde.org.

(a)

(b)

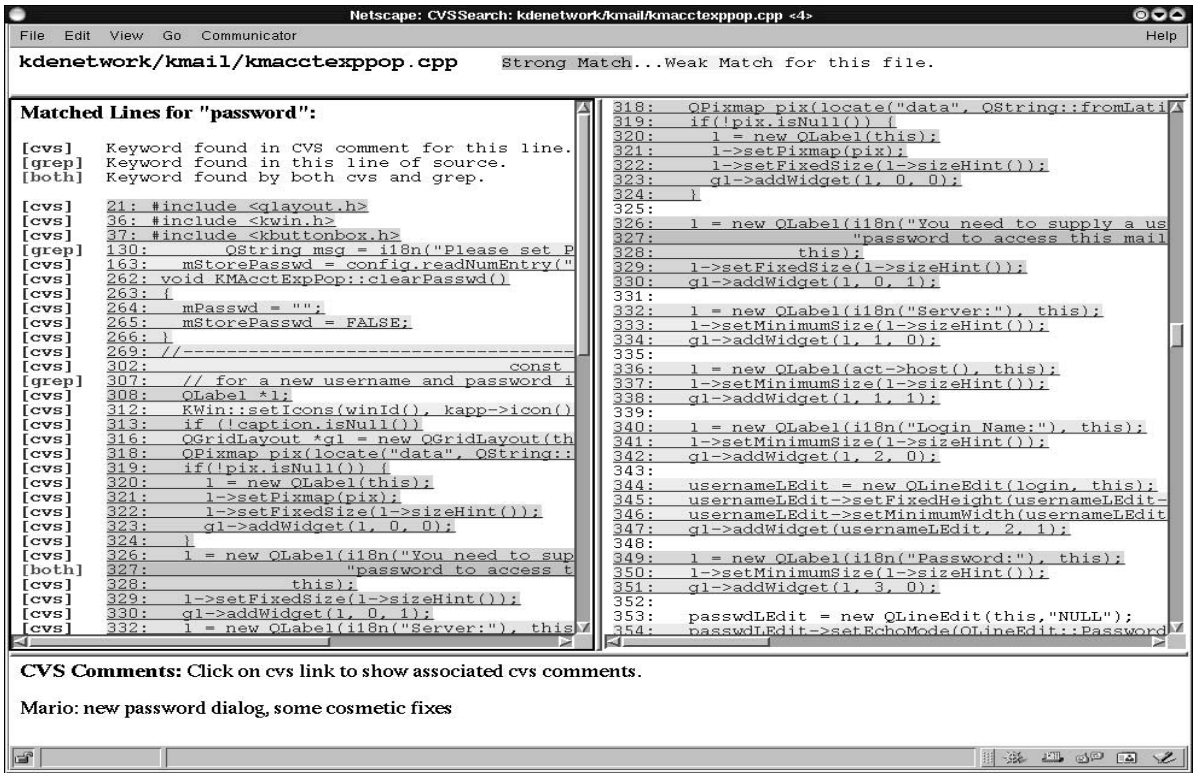


Figure 2. CVSSearch Screenshots.

KDE Application	Lines of Code	Average # Rev./File	DB Build (min.)	DB Space (KB)	Average # Comments/Line
konqueror	24,253	38.3	23	2,911	1.5
korganizer	43,188	10.8	9	5,028	1.2
kmail	40,325	32.9	24	3,672	1.4
knode	33,721	14.7	8	2,237	1.3
kword	48,883	22.5	47	4,475	1.6

Figure 3. CVSSearch statistics for KDE applications. (Timing was done on a 700Mhz Pentium III with 256 MB.)

	CVSSearch	grep	same	Total	CVSSearch better	grep better
					p-value	p-value
konqueror	57 (40%)	41 (29%)	44 (31%)	142 (100%)	0.104	0.923
kmail	63 (46%)	38 (28%)	37 (27%)	138 (100%)	0.0249	0.984
knode	53 (38%)	48 (34%)	39 (28%)	140 (100%)	0.400	0.664
korganizer	41 (29%)	68 (48%)	32 (23%)	141 (100%)	0.991	0.0141
kword	70 (49%)	29 (20%)	43 (30%)	142 (100%)	0.000497	0.9997
Total	284 (40%)	224 (32%)	195 (28%)	703 (100%)	0.0143	0.988

Table 1. Results of query evaluation.

Another wrote "CVS comments (are) often an improvement over much of the open source code commenting I've seen... allows for longitudinal (time wise) searching, e.g., bug-fixes, recent areas of development." Other comments along this direction: "good for getting a perspective on what works and what's under development, and locating these points quickly" and "CVSSearch provides more comments behind why the code was written".

The CVS comments can themselves be used to explain matches (as we do now using CVSSearch). Regarding this point, one person wrote "CVS comments associated with the matched files were very helpful in not only understanding and determining relevance of code but also showing the role of the code in the entire program."

We see that CVS comments do provide a valuable source of information that complements — but does not replace — content-based matching (e.g., using grep). Moreover, CVS comments are also good at explaining the lines matched, and indeed, can be used as an additional source of documentation for code irrespective of search.

5 Related Work

There is a myriad of search tools for code, some of which search through code directly [8, 16, 22], and others that search by looking at the natural-language documentation (such as comments or manual pages) associated with the code [12, 19].

In the first category, we find lexical tools such as grep [16], awk [1], lex [18], perl [23], and LSME [21]. Such tools are based on regular expressions, and while simple to use, have problems searching for certain constructs. For example, grep is not designed to search for patterns spanning multiple lines. Moreover, regular expressions are quite limited. For example, it is not possible to search for two statements at the same level of nesting. Consequently, search tools have been developed that parse the source code [8, 15, 17, 22] to alleviate such problems.

As discussed in Section 1, tools of this type, whether lexical or syntactic, are quite difficult to use if the user is not familiar with the code at all. This is particularly an issue when the code is poorly commented since the query words would no longer likely match words in the natural-language code comments.

In the second category, we find tools that are based on natural-language documentation associated with the code [4, 6, 11, 12, 19]. It is worthwhile noting that such tools are designed for retrieval of reusable components whereas CVSSearch is a general purpose search tool. Moreover, as far as we know, CVSSearch is the only search tool based on CVS comments — which not only have the advantages discussed in Section 1 — but are often the only source of documentation in open source projects.

Research has been done that uses version control system history in various ways: evaluating the impact of software tools [5]; inferring change effort [14]; predicting fault incidence [13]; and detecting code decay [9]. As far as we know, CVSSearch is the only tool that makes use of version

control system history to provide a general purpose search facility.

6 Conclusions and Future Work

In this paper, we have introduced a method for finding fragments of source code by using CVS comments. Our approach takes advantage of the fact that a CVS comment describes the lines of code involved in the commit and this description will typically hold for many future versions.

We have presented an algorithm for associating CVS comments with lines in the most recent version of the code. Moreover, we described our tool, CVSSearch, that is based on this algorithm and which we have used on real-life KDE applications of significant size. Preliminary evaluation shows that CVS comments do provide a valuable source of information that complements — but does not replace — content-based matching (e.g., using grep).

On possible avenue for future research is to provide a program understanding feature that allows users to select arbitrary lines in the code (which need not appear in a contiguous block) and CVSSearch will provide an explanation of what those lines do *collectively*. This can be done by looking at the CVS comments associated with each of the selected lines and combining them in some interesting way. For example, we could eliminate duplicates and rank the comments in order of their prevalence in the profiles of the selected lines. In this way, the comments shown first are likely to apply to many of the lines selected by their user — and are thus likely to indicate a common role played by those lines.

Our ultimate goal is to provide a search engine based on CVSSearch to the open source community whose database encompasses a large number of open source projects. Not only should one be able to search individual projects but also all of them at once. Such a feature can be useful in learning to use a software library. As an example, one may be interested in knowing how people do double buffering using the KDE core libraries. Of course, we could also mine the results of the search to look for patterns, much as is done in the CodeWeb tool [20] (e.g. we could find patterns in the way people do double buffering using the KDE core libraries). Since this paper was submitted, we have started such work by adding a data mining module in CVSSearch to provide such functionality (e.g. searching for double buffering returns relevant classes and functions such as QPixmap and drawPixmap()).

References

[1] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. Awk — a pattern scanning and processing language. *Software Practice and Experience*, 9(4):267–280, 1979.

[2] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia. Information retrieval models for recovering traceability links between code and documentation. In *Proceedings of the International Conference on Software Maintenance*, pages 40–49, 2000.

[3] G. Antoniol, G. Canfora, and A. De Lucia. Recovering code to documentation links in OO systems. In *Proceedings of the Working Conference on Reverse Engineering*, pages 136–144, 1999.

[4] S. P. Arnold and S. L. Stepoway. The reuse system: Cataloging and retrieval of reusable software. In *Software Reuse: Emerging Technology*, pages 138–141, 1987.

[5] D. Atkins, T. Ball, T. Graves, and A. Mockus. Using version control data to evaluate the impact of software tools. In *International Conference on Software Engineering*, pages 324–333, 1999.

[6] B. A. Burton, R. W. Aragon, S. A. Bailey, K. D. Koelher, and L. A. Mayes. The reuse system: Cataloging and retrieval of reusable software. In *Software Reuse: Emerging Technology*, pages 129–137, 1987.

[7] A. Chen, Y. K. Lee, A. Y. Yao, and A. Michail. Code search based on CVS comments: A preliminary evaluation. Technical Report UNSW CSE TR 0106, University of New South Wales, 2001.

[8] P. Devanbu. GENOA — a customizable, language and front-end independent source code analyzer generator. In *Proceedings of the 14th International Conference on Software Engineering*, pages 307–317, 1992.

[9] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[10] K. F. Fogel. *Open Source Development with CVS*. Coriolis Inc., 2000.

[11] W. B. Frakes and B. A. Nejmeh. Software reuse through information retrieval. In *20th Hawaii International Conference on System Sciences*, pages 530–535. IEEE, 1987.

[12] M. R. Girardi and B. Ibrahim. Using English to retrieve sofware. *The Journal of System and Software*, 30(3):249–270, 1995.

[13] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[14] T. L. Graves and A. Mockus. Inferring change effort from configuration management databases. In *Proceedings of the 5th International Symposium on Software Metrics*, pages 267–272, 1998.

[15] W. G. Griswold, D. C. Atkinson, and C. McCurdy. Fast, flexible, syntactic pattern matching and processing. In *Proceedings of the 4th Workshop on Program Comprehension*, pages 144–153, 1996.

[16] B. W. Kernighan and R. Pike. *The Unix Programming Environment*. Prentice Hall, 1984.

[17] D. A. Ladd and J. C. Ramming. A*: A language for implementing language processors. *IEEE Transactions on Software Engineering*, 21(11):894–901, 1995.

[18] M. E. Lesk. Lex — a lexical analyzer generator. Technical report, AT&T Bell Laboratories, Murray Hill, N. J., 1975.

[19] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813, 1991.

[20] A. Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the 22nd International Conference on Software Engineering*, 2000.

[21] G. C. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering and Methodology*, 5(3):262–292, 1996.

[22] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, 1994.

[23] L. Wall. *Programming Perl*. O'Reilly and Associates, 1990.

[24] M. S. Waterman. *Introduction to Computational Biology: Maps, Sequences, and Genomes*. CRC Press, 1995.

[25] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes*. Morgan Kaufmann, 1999.

Appendix: Alignment Algorithm

String Alignment

In this section, we use $|S|$ to denote the length of a string S, and we use $S[i]$ to denote the ith character of S, where the first character is denoted by S_1 (rather than S_0).

If S and T are strings, then an *alignment* A maps S and T into strings S' and T', possibly containing insertion of "spaces", such that $|S'| = |T'|$ and the removal of these spaces from S' and T' leaves S and T, respectively. (These "spaces" are denoted by '-'; they must be distinct from any characters in the strings.) For example, given two strings acbcdb and cadbd, one possible alignment of these two strings is:

```
a c - - b c d b
- c a d b - d -
```

To determine the quality of an alignment, we use a scoring function. For example, if an exact match between two characters scores +2 and every mismatch scores -1, then the alignment above has score $3 \times (+2) + 5 \times (-1) = 1$.

If x and y are each a single character or space, then $\sigma(x, y)$ denotes the *score* of aligning x and y. The *value* of an alignment A of S and T is $\sum_{i=1}^{l} \sigma(S'[i], T'[i])$, where $l = |S'| = |T'|$. An *optimal alignment* of S and T is one that has the maximum possible value for these two strings.

One can use a simple dynamic programming algorithm to compute the optimal alignment of two strings S and T in time $\Theta(|S| \times |T|)$ and space also $\Theta(|S| \times |T|)$ [24].

Source Code Alignment

To identify lines in one version of a file that are "similar" to lines in a previous version of that file, we compute an alignment of the two versions of the file. We then identify lines one version that should be mapped with lines in the previous version. The mapping is one-to-one in the sense that a line in one version is mapped to at most one line in the other version and vice versa.

The alignment of the two files is done using string alignment at two levels. At one level, we consider strings F_1 and F_2, where F_i is the sequence of lines in file version i. (That is, each "character" in F_i actually corresponds to the line in file version i.) The scoring function $\sigma(F_1[i], F_2[j])$ is equal to the value of the optimal alignment of lines $F_1[i]$ and $F_2[j]$. This latter alignment of two lines, consisting of sequences of characters, is done using a scoring function that rewards character matches with +2 and mismatches with -1.

GNU Diff

The GNU diff command can also be used to perform alignments of two versions of a file, though it does not look at how similar lines are across two files. Rather, it simply checks whether two lines are the same or not (modulo changes in whitespace if the -b flag is used). However, our alignment algorithm actually performs another lower level alignment to determine the similarity of lines.

The GNU diff command is faster than our algorithm, and so we use it to come up with a initial alignment of the two file versions. However, whenever the GNU diff command reports that a contiguous block of lines in one version corresponds to another contiguous block of lines in another version (e.g, $s_1, s_2\, c\, d_1 d_2$) — which does not tell us precisely which lines in one block correspond to which lines in the other block — we use our alignment algorithm on the two blocks to determine this information.