Toward Mining "Concept Keywords" from Identifiers in Large Software Projects

Masaru Ohba
Tokyo Institute of Technology
2-12-1 Oookayama Meguro
Tokyo 152-8552, JAPAN
m-ohba @sde.cs.titech.ac.jp

Katsuhiko Gondow
Tokyo Institute of Technology
2-12-1 Oookayama Meguro
Tokyo 152-8552, JAPAN
gondow @cs.titech.ac.jp

ABSTRACT

We propose the Concept Keyword Term Frequency/Inverse Document Frequency (ckTF/IDF) method as a novel technique to efficiency mine *concept keywords* from identifiers in large software projects. ckTF/IDF is suitable for mining concept keywords, since the ckTF/IDF is more lightweight than the TF/IDF method, and the ckTF/IDF's heuristics is tuned for identifiers in programs.

We then experimentally apply the ckTF/IDF to our educational operating system `udos`, consisting of around 5,000 lines in C code, which produced promising results; the `udos`'s source code was processed in 1.4 seconds with an accuracy of around 57%. This preliminary result suggests that our approach is useful for mining concept keywords from identifiers, although we need more research and experience.

Keywords

concept keywords, program understanding, identifiers, TF/IDF

1. INTRODUCTION

Many programmers make all possible effort to make their identifiers both concise and descriptive enough to suggest their role in a program (for example, "`read_dirent()`" in `udos`[10], as opposed to "`f()`"). Fortunately, such descriptive identifiers provide a wealth of information which can aid in program understanding. From the previous example, `dirent` implies "directory entry" - a key concept in understanding the FAT file system[2]. Thus, we see the benefits for program understanding that mining such terms can bring. In this paper we present a tool which can mine such key concepts, called *concept keywords*, from identifiers present in source code.

Concept keywords, for the most part, contribute to program understanding in three ways.

- Concept keywords highlight important parts in source code and implicit relations among them.

  In program understanding, not all code fragments are equally important, and important parts are unequally distributed in source code. Also important parts change depending on your concern. By using, for example, text editor highlighting[1] for concept keywords in your concern (e.g., `dirent`), you would quickly find important parts in source code.

  Also concept keywords help you to find implicit relations in source code. For example, by searching `dirent`, you can find a comment like "`/* FAT12_read` is used for sequential access to directory entries, while `read_dirent` for random access `*/`". This relation between `read_dirent` and `FAT12_read` is implicit in the sense that there is no control dependence nor data dependence between them.

- Concept keywords bridge the gap in understanding between source code and specifications, caused by abstraction mismatch.

  By relating the parts with a same concept keyword, you can find the corresponding descriptions more efficiently. For example, when you find a function "`read_dirent`" in source code, you can imagine the function reads a "directory entry", and know its structure by searching "directory entry" and "dirent" in FAT specification[2]. Of course concept keywords do not work well for vocabulary mismatch (e.g., the function name `read_folder_entry` for reading a directory entry); we assume most programmers try to avoid such vocabulary mismatch in naming identifiers.

- Identifiers have an good affinity for software repositories and a technique for mining concept keywords can also be applied to large software repositories.

  This is because the characteristics of a typical software repository such as the combination of version control system, mailing list system and bug tracking system are almost the same as those of identifiers; both of them are language independent, text-based, machine-processable in a lightweight manner, and so on.

Many program understanding tools are already available such as call-graph extractors, cross-referencers, slicers, outlining tools, source code browsers, beautifiers, code metrics tools, documentation tools, debuggers, profilers, etc. However, none of these tools can mine concept keywords. Our goal is to develop a novel tool for mining concept keywords from identifiers.

Unfortunately, it is very challenging to efficiently and accurately mine concept keywords from identifiers in large software since concept keywords are hidden within numerous identifiers in a somewhat "unexpected" manner; thus we need to use a lightweight heuristic mining algorithm suitable for identifiers. Existing and well-known mining algorithms such as the TF/IDF weighting method[16] does not work well for identifiers, since the characteristics of concept keywords and identifiers are quite different from those of natural languages. For example, identifier prefixes such as `kbd_` (meaning "keyboard"), are often used to group strongly related identifiers, but this does not occur in natural language. The TF/IDF method gives high scores to prefixes which are not concept keywords, resulting in inaccurate mining.

[1] For example, `highlight-regexp.el` for the Emacs editor.

MSR '05 Saint Louis, Missouri USA

Moreover, existing algorithms may prove too heavy for mining concept keywords. Software is continually getting larger and more complex - GCC (GNU Compiler Collection) has over 400,000 lines in C, and requires over 30 minutes to build[2]. Furthermore, reducing the time-to-market is of paramount importance in today's software development projects. Hence programmers require efficient mining tools and need to find a new technique suitable for mining concept keywords.

In this paper we propose the Concept Keyword Term Frequency and Inverted Document Frequency (ckTF/IDF) method to efficiently and accurately mine concept keywords. Our basic ideas are:

- ckTF/IDF is a very lightweight method, obtained by simplifying TF/IDF scoring.
- ckTF/IDF is tuned for identifiers. For example, ckTF/IDF excludes meaningless prefixes with a high accuracy.

To see how this idea works in practice, we experimentally applied ckTF/IDF method to our educational operating system `udos`[10] (about 5,000 lines in C). `udos` is selected as a testbed, since:

- We are familiar with the `udos` source, since one of the authors (Gondow) developed `udos`. Thus we can examine the result of the experiment using our knowledge about `udos`.
- Concepts in operating systems can be enumerated from OS text books and specifications such as POSIX, hardware manuals, etc.
- `udos` is a relatively small operating system, yet complicated enough to realize the importance of concept keywords.

As a result, ckTF/IDF processed `udos`'s source code in 1.4 seconds with an accuracy of around 57%. This preliminary result suggests that our approach is helpful for mining concept keywords from identifiers, although we need more research and experience.

This paper is organized as follows. Section 2 describes the characteristics of concept keywords, and the difficulty of mining them. Section 3 introduces our new ckTF/IDF method. Section 4 explains our experimental implementation of the framework for the ckTF/IDF. Section 5 describes our preliminary experiment of applying ckTF/IDF to `udos` and its results. Section 6 describes related work. Section 7 gives our conclusions and suggestions for future work.

2. CONCEPT KEYWORDS

This section describes the characteristics of concept keywords, and the benefits and difficulty of mining them.

2.1 What is a concept keyword?

In Section 1, we mentioned a concept keyword is a word that represents a key concept in program understanding, and `dirent` (directory entry) is an instance of concept keywords. So what is the definition of concept keywords? Unfortunately there is no clear definition of concept keywords, since they are highly based on subjective judgement. To make our discussion clear, we use the following three terms for concept keywords.

- *Ideal concept keywords*, which have proven to improve program understanding by some objective measurements.
- *Human-selected concept keywords*, which a developer or reviewer believes are ideal concept keywords.
- *Machine-extracted concept keywords*, which a method like TF/IDF produced as an approximation of ideal or human-selected ones.

[2]By GCC-3.4.3 on Pentium 4(supported HT) 2.6GHz, 512MB RAM, Linux-2.6.8-1-686-smp

For example, `dirent` is a human-selected concept keyword in the sense that we just judged so. Although it is still unknown due to the lack of good metrics whether `dirent` is an ideal one or not, our software development experience suggests a hypothesis that concept keywords exist as a small subset of words in identifiers.

Table 1 shows the number of all human-selected concept keywords that we selected from `udos`'s source code, along with those of words in other categories. Table 1 tells around 22% ($= 61/279$) of words in identifiers are human-selected concept keywords, which supports the above hypothesis.

Hereafter, in this paper, we often use the term *concept keyword* for referring to a *human-selected concept keyword*. In Section 5, *machine-extracted concept keywords* by ckTF/IDF and TF/IDF are compared with *human-selected concept keywords* in Table 1.

2.2 Why concept keywords?

In this section, we explain why mining concept keywords from identifiers has a great potential to dramatically improve program understanding.

Program understanding is the most important activity in software development and software maintenance. In [12], for example, it is estimated that "some 30-35% of total life-cycle costs are consumed in trying to understand software after it has been delivered, to make changes". Thus improving program understanding is a key issue in software engineering.

To alleviate this problem, as mentioned in Section 1, many program understanding tools have already been developed and researched such as call-graph extractors[11], cross-referencers, slicers, outlining tools, etc. Although many successes have been achieved in individual areas by these tools, the cost of program understanding still remains high.

This reason is twofold. One reason is inaccuracy in tools. An empirical study [14], for example, reported that call graphs extracted by several broadly distributed tools vary significantly enough to surprise many experienced software engineers. The other reason is the limitations of tools. For example, even ideal call-graph extractors cannot cover the whole range of information required in program understanding. This observation leads us to the necessity of developing yet another kind of tools, and of discovering some ways to well combine them with the existing tools.

Our idea of mining concept keywords provides a new kind of support for program understanding. As mentioned in Section 1, concept keywords can contribute to programing understanding in various ways. Also it is easy to combine our mining technique with the existing tools like version control system, mailing list system and bug tracking system, since both of them have the same characteristics of being language independent, text-based, machine-processable in a lightweight manner. Thus, the idea of mining concept keywords seems very attractive and can have a great potential to improve program understanding. As far as we know, however, little work has been done so far for mining concept keywords in source code.

2.3 Why difficult to mine concept keywords?

As mentioned in Section 1, however, existing and well-known mining algorithms such as the TF/IDF weighting method do not work well for identifiers, since the characteristics of concept keywords and identifiers are quite different from those of natural languages, and since existing algorithms can be too expensive for mining concept keywords.

In order to accurately mine concept keywords, we feel it is necessary for users to experiment with parameters (such as the term frequency threshold) - hence, a lightweight algorithm is preferred over a more expensive one in order to enable frequent experimentation.

Hence, we intrude ckTF/IDF, an algorithm based on TF/IDF which is less expensive without sacrificing quality of keywords mined. The differences between the two algorithms will be shown in Section 3.3 and 5.

Table 1: Human-selected concept keywords and other category words in udos

	category	#	examples	description
1.	concept keywords	61	`dirent`, `root`, `PTE`, `tss`, `path`, `signal`, `yield`	helpful key concepts for program understanding
2.	grouping words	18	`kbd_`, `vga_`, `FAT12_`, `sys_`, `FDC_`, `RTC_`, `console_`, `_H`, `_t`	prefixes and suffixes for grouping functions and variables, or for other purposes
3.	attributes, and less important concepts	70	`busy`, `byte`, `offset`, `name`, `memory`, `end`, `int8`, `again`	general nouns and adjectives used as attributes, modifiers, etc., being less informative in themselves.
4.	generic verbs	130	`read`, `set`, `is`, `move`, `wait`, `print`, `dump`, `make`, `init`	generic verbs to describe actions or operations; the same names are commonly used for unrelated functions

3. ckTF/IDF METHOD

In this section, we propose the *Concept Keyword Term Frequency and Inverted Document Frequency* (ckTF/IDF) method to efficiently mine concept keywords.

3.1 Overview of TF/IDF method

Before we define the ckTF/IDF method, this section gives an overview of the TF/IDF method, which ckTF/IDF is based on. The TF/IDF is a method for mining characteristic terms and document classification. The key feature of TF/IDF heuristics is to give a high score (i.e., high term weight) as a characteristic term if the term appears more frequently in a particular document and less in other documents. The weight of a term in a document is calculated by its term frequency (TF) and its inverse document frequency (IDF) in all the documents.

The inverse document frequency $idf(t)$ for a term t is defined as follows.

$$idf(t) = \log \frac{N}{df(t)} \tag{1}$$

where $df(t)$ is the number of documents including the term t, N the number of all documents.

The term weight $w(t, d)$ for a term t and a document d is defined as follows.

$$w(t, d) = tf(t, d) \cdot idf(t) \tag{2}$$

where $tf(t, d)$ (meaning the term frequency) is a count of occurrence of term t in document d.

Thus keywords can be mined from documents in natural languages by picking terms with high weights calculated by TF/IDF.

3.2 Definition of ckTF/IDF method

This section gives the definition of the ckTF/IDF method. The ckTF/IDF treats one source file as one document, since most source code written in C are expedient granularity for mining concept keywords from our experience. For example, `dirent` and `root` of concept keywords in FAT file system appear in `udos`'s `fat12.c` only, and not appear in others.

The ckTF/IDF is a very simplified and thus speeded-up version of TF/IDF by quantizing $tf(t, d)$ and $idf(t)$ into 0 or 1. The $idf(t)$ for ckTF/IDF is defined as follows.

$$idf(t) = \begin{cases} 1 & \text{if } 1 \leq df(t) \leq n \text{ and } \neg\text{is\_prefix}(t) \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

where $df(t)$ is the same as in TF/IDF, $n(\geq 1)$ the threshold (default is 1) to quantize $tf(t, d)$ and $idf(t)$, and is_prefix(t) a predicate being true if and only if t is a prefix for all its occurrences.

The term frequency $tf(t)$ in all documents and the word weight $w(t)$ for ckTF/IDF are defined as follows.

$$tf(t) = \begin{cases} 1 & \text{if } \exists d, tf(t, d) > n \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

$$w(t) = tf(t) \cdot idf(t) \tag{5}$$

In ckTF/IDF, $w(t) = 1$ implies the term t is characteristic, so t is selected as a (machine-extracted) concept keyword.

The value of $w(t)$ can be computed very fast by using the two flags: *local frequency flag* and *global frequency flag* for each term t (*local*(t) and *global*(t) for short, respectively). First all identifiers in source code are divided into terms by some delimiters like underscores. Then two flags are computed for each term, considering a language construct for grouping (e.g., a compilation unit for the programming language C) as a document. When a term t is found twice in a document, *local*(t) is set without performing the remaining computation. Similarly, when t is found in two documents, *global*(t) is set without performing the remaining computation. Note that *local*(t) and *global*(t) are not exclusive; both can be set at the same time. If *local*(t) is set, *global*(t) is clear, and the term is not a prefix, then $w(t)$ becomes 1. Otherwise, $w(t)$ becomes 0.

Thus ckTF/IDF realizes a lightweight way of mining concept keywords. The computational complexity of ckTF/IDF and TF/IDF is discussed in Section 3.3. Some actual measurements of their performance are shown in Section 5.

3.3 ckTF/IDF vs. TF/IDF

3.3.1 Characteristics of ckTF/IDF

For most cases, $w(t) = 1$ for ckTF/IDF when $\max_d w(t, d)$ for TF/IDF is high, and $w(t) = 0$ when $\max_d w(t, d)$ is low. Thus there is a high correlation between ckTF/IDF and TF/IDF. There are two exceptions for this.

- ckTF/IDF imposes a penalty for prefix terms, while TF/IDF not. This penalty is introduced to ckTF/IDF, since prefixes like `kbd_` (meaning "keyboard") are not likely to be concept keywords from our experience, and also since this penalty can be computed fast.

- When two flags are set at the same time, ckTF/IDF's score is always 0, while TF/IDF's score varies. This can be the cause of inaccuracy of ckTF/IDF, when $tf(t, d)$ is very large or $df(t)$ is small but greater than 1. This can be alleviated by adjusting the threshold n in Equation (3) and (4).

3.3.2 Computational complexity

For computing all $w(t)$ and $\max_d w(t, d)$ from the scratch, computational complexity of both ckTF/IDF and TF/IDF are the same as $O(|D|+|T|)$ where D is a set of all documents and T a set of all terms, although ckTF/IDF is much faster than TF/IDF in practice (see also Section 5).

The difference arises when computing them incrementally. Suppose that we add a set of documents ΔD including a set of terms ΔT. The complexity for ckTF/IDF is $O(|\Delta D| + |\Delta T|)$, while that for TF/IDF is $O(|D + \Delta D| + |T + \Delta T|)$.

4. DESIGN AND IMPLEMENTATION

This section gives a brief description of the design and implementation of Identifer Exploratory Framework (IEF), which we experimentally developed as a framework for the ckTF/IDF method (IEF's source code is available in [15]). Figure 1 shows the overview

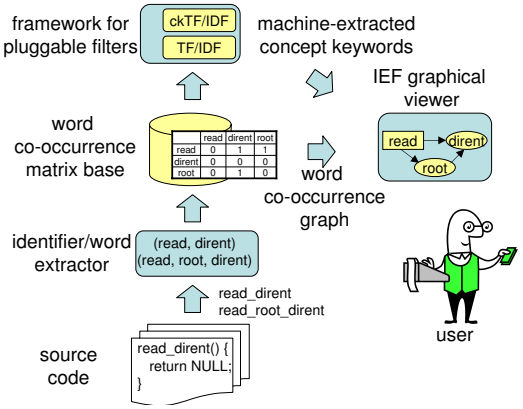


Figure 1: Components of Identifier Exploratory Framework (IEF)

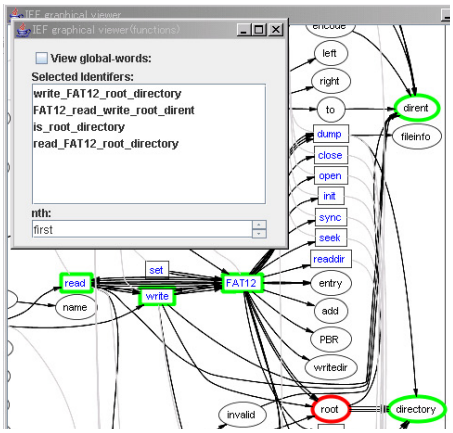


Figure 2: Screen snapshot of IEF graphical viewer

of IEF. Figure 2 shows a sample screen snapshot of IEF's GUI, which displays a co-occurrence graph for words in `udos`'s `fat12.c`. The components of IEF are:

- *Identifier/word extractor* - extracts all definitions of function and global variables from a given source code by utilizing a cross-referencer GNU GLOBAL[3], and then tokenizes their names into words. Note that uses of functions/variables are not extracted to avoid the increase of document frequency for important global functions. It simply uses underscore (_) as delimiter to avoid heavyweight processing like morphological analysis or use of dictionary. Thus it is lightweight, although it cannot tokenize some identifiers like `kmalloc` (meaning "kernel memory allocation"). It consists of around 300 lines in Ruby script language[5] and some C code for GNU GLOBAL.

- *Word co-occurrence matrix base* (i.e., word corpus): is a simple database that stores all words extracted from identifiers, along with co-occurrence information in identifiers and their filenames.

- *Framework for pluggable filters*: is the core feature of IEF. It provides the extensibility for implementing additional filters, and also provides interface for filters to access the word co-occurrence matrix. Currently only the filters for ckTF/IDF and TF/IDF are available, which consist of around 400 lines in Ruby.

- *IEF graphical viewer*: allows users to browse the word co-occurrence graph and the output of the filters. IEF graphical viewer is implemented using Grappa graph-drawing library[4]. It consists of around 500 lines in Java.

Table 2: Total # of word occurrences by position for `udos`

	category	word position		
		first	last	middle
1.	concept keywords	*14*	21	75
2.	grouping words	*141*	20	33
3.	attributes	78	17	199
4.	generic verbs	23	38	41
	total	*256*	96	348

Figure 2 shows a screen snapshot of IEF graphical viewer, which displays a co-occurrence graph of words in `fat12.c` of `udos`. In the graph, a rectangle node implies a term t such that $global(t) = 1$, an oval node implies a term t such that $global(t) = 0$, and an edge implies the two end nodes of the edge co-occurs in a same identifier. At the first time we saw the co-occurence graph, we soon found that human-selected concept keywords like `dirent` or `root` almost correspond to non-global nodes with many edges in the graph. This experience actually motivated us to start this research.

5. PRELIMINARY EXPERIMENT

To see how ckTF/IDF works in practice, we experimentally applied the ckTF/IDF and TF/IDF to our educational operating system `udos`'s source code[10]. This section gives the results of this experiment.

5.1 Accuracy and Coverage of ckTF/IDF

In this paper, we use the measures of accuracy and coverage to evaluate the performance ckTF/IDF and TF/IDF by comparing the concept keywords extracted by a human programmer with that of the algorithm. Accuracy and coverage correspond to a measure of precision and recall, respectively. We define $Accuracy = \frac{C_r}{C_m}$ and $Coverage = \frac{C_r}{C_h}$. where C_m is the total number of all machine-extracted concept keywords, C_h is the total number of human-extracted concept keywords , and C_r is the total number of concept keywords that both human and algorithm have chosen.

Figure 3 shows accuracy and coverage when applying ckTF/IDF the the codebase of `udos`. We found an accuracy of 57% (=16/28) and a coverage of 26% (=16/61) and that the accuracy in mining concept keywords increases by removing uncecessary words, those which are not considered to be of less significance as defined in "(c)" and "(d)" in Figure 3.

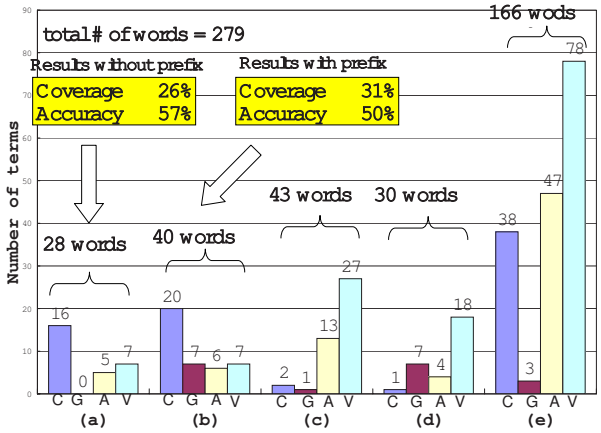
In contrast, the accuracy and coverage of TF/IDF for `udos` are 28% (=8/28) [3] and 13% (=8/61), respectively (not shown in any figure). Thus, as far as this experiment is concerned, ckTF/IDF delivers twice better accuracy and coverage than TF/IDF.

5.2 Removing Prefixes Improves Accuracy?

In Section 3.3.1, we mentioned ckTF/IDF imposes a penalty for prefix terms under the hypothesis that prefixes are unlikely to be concept keywords. Table 2 supports this hypothesis, which tells that concept keywords rarely occur in the first position (around 5% = 14/256), while grouping words often occur there (around 55% = 141/256).

So does the heuristics of ckTF/IDF work? The answer is subtle. Figure 3's (a) and (b) explain this subtlety. (a) is the result with the prefix penalty, while (b) is the result without it. By using the prefix penalty, the accuracy increased to 57% (7% up), but the coverage decreased to 26% (5% down).

[3]TF/IDF outputs word weighting for all entries as results, while ckTF/IDF outputs only candidates of concept keywords. Therefore, in order to compare ckTF/IDF's results with that of TF/IDF, we limited the comparison to the first n candidates, where n is the number of candidates returned by ckTF/IDF.



Legend: C=concept keywords, G=grouping words, A=attributes, V=verb,
(a)= $\neg global(t) \wedge local(t) \wedge \neg$is_prefix$(t)$
(b)= $\neg global(t) \wedge local(t)$, (c)= $global(t) \wedge \neg local(t)$
(d)= $global(t) \wedge local(t)$, (e)= $\neg global(t) \wedge \neg local(t)$

Figure 3: Accuracy and coverage of ckTF/IDF for `udos`

5.3 Performance of ckTF/IDF and TF/IDF

As mentioned before, ckTF/IDF processed `udos`'s source code in 1.4 seconds (including file I/O time). `udos` (around 5,000 lines in C), however, is too small to compare the performance, so we used the Ruby interpreter instead of `udos`. Ruby consists of around 48,000 lines in C (excluding blank lines), and has 3,385 identifiers.

Table 3 shows a comparison of the execution speeds[4] of ckTF/IDF and TF/IDF for the Ruby interpreter. "Computation from scratch" in Table 3 means the term weight computation of the whole source code of the Ruby intepreter, and "incremental computation" means re-computation of the above results after a document including 7 words is added. As far as the results are concerned, ckTF/IDF is about 6 times faster than TF/IDF. Note that "0 sec" in Table 3 means a very small amount of time, not really zero.

6. RELATED WORK

To our knowledge, little work has been done so far for mining concept keywords in program identifiers.

Caprile et al.[8, 9] proposes an identifier restructuring tool, which uses a semiautomatic technique for the restructuring of identifiers, and enforces a standard syntax for their arrangement. They consider identifiers as an important tool for programming understanding, but their research is not for mining concept keywords in program identifiers.

Anquetil[6] attempts manually mining concepts from program identifiers and comments. He applied his manual technique to the Mosaic system, relating, for example, `xm` and `xmx` to the X Window System. His experiment took quite a long time (around 30 hours), while our mining by ckTK/IDF took only around 1.4 seconds to automatically process `udos`'s source code.

Anquetil et al.[7] proposes a technique for extracting concepts from the abbreviated filenames (such as "dbg" for debug or "cp" for call processing). Although they achieved a high accuracy (80% to 85% of abbreviations found when used with an English dictionary), their technique seems too heavy for mining identifiers in large software projects.

Knuth[13] tries to achieve better program understanding by integrating both of source code and documents using the `WEB` language. In contrast, our aim is to achieve better program understanding for large open source code like GCC, not written in `WEB`, by mining concept keywords from program indentifiers.

7. CONCLUSION

[4]File I/O time is excluded.

Table 3: Execution speeds of ckTF/IDF and TF/IDF for Ruby interpreter (in elapsed time)

	ckTF/IDF	TF/IDF
computation from scratch	0.24 sec	1.57 sec
incremental computation	0 sec	0.44 sec

We have proposed the Concept Keyword Term Frequency/Inverse Document Frequency (ckTF/IDF) method as a novel technique to efficiency mine *concept keywords* from identifiers in large software projects. Testing the algorithm using the source code of `udos` (5,000 lines of C), we found that it was processed in 1.4 seconds with an accuracy of around 57% and coverage of around 26%. Coverage of around 26% is not necessarily high, although the ckTF/IDF method is extremely lightweight and high in accuracy. We assume that the coverage can be increased by incorporating approaches used in other mining algorithms.

This preliminary result suggests that our approach is helpful for mining concept keywords from identifiers, although we need more research and experience.

Our future works include: (1) to apply our mining technique to large practical software like GCC or Apache and to provide comprehensive evaluation, (2) to apply concept keywords and/or the ckTF/IDF method to a Bug Tracking System (BTS) like bugzilla[1] to relate keywords in bug reports to the corresponding source code,

8. REFERENCES

[1] Homepage for bugzilla. `http://bugzilla.mozilla.org/`.

[2] Homepage for FAT32 file system specification. `http://www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx`.

[3] Homepage for GNU GLOBAL. `http://www.gnu.org/software/global`.

[4] Homepage for Grappa. `http://www.research.att.com/~john/Grappa/`.

[5] Ruby language homepage. `http://www.ruby-lang.org`.

[6] Nicolas Anquetil. Characterizing the informal knowledge contained in systems. In *WCRE: Proc. 8th Working Conf. on Reverse Engineering*, pages 166–175, 2001.

[7] Nicolas Anquetil and Timothy Lethbridge. Extracting concepts from file names: a new file clustering criterion. In *ICSE '98: Proc. 20th Int. Conf. on Software Engineering*, pages 84–93. IEEE Computer Society, 1998.

[8] Bruno Caprile and Paolo Tonella. Nomen est omen: Analyzing the language of function identifiers. In *WCRE '99: Proc. 6th Working Conf. on Reverse Engineering*, page 112. IEEE Computer Society, 1999.

[9] Bruno Caprile and Paolo Tonella. Restructuring program identifier names. In *ICSM: Int. Conf. on Software Maintenance*, pages 97–107, 2000.

[10] K. Gondow. Homepage for an educational operating system `udos`. `http://www.sde.cs.titech.ac.jp/~gondow/udos/`.

[11] K. Gondow, T. Suzuki, and H. Kawashima. Binary-level lightweight data integration to develop program understanding tools for embedded software in c. In *Proc. 11th Asia-Pacific Software Engineering Conference (APSEC)*, pages 336–345, 2004.

[12] P. A. V. Hall. Overview of reverse engineering and reuse research. *Information and Software Technology*, 34(4):239 – 249, 1992.

[13] Donald E. Knuth. *Literate Programming (Center for the Study of Language and Information - Lecture Notes, No*. Van Nostrand Reinhold Computer, 1989.

[14] G.C. Murphy, D. Notkin, and E.S.-C. Lan. An empirical study of static call graph extractors. In *Proc. 18th Int. Conf. on Software Engineering (ICSE-18)*, pages 90–99, 25–29 Mar 1996.

[15] M. Ohba. Homepage for the concept keyword mining tool. `http://www.sde.cs.titech.ac.jp/~m-ohba/cktfidf/`.

[16] Gerard Salton and Christopher Buckley. Termweighting approaches in automatic text retrieval. *Information Processing and Management*, Vol. 24(5), 1988.