Code Similarity Comparison of Multiple Source Trees

Dr. Warren Toomey,
Assistant Professor
School of IT, Bond University
wtoomey@staff.bond.edu.au

April 23, 2008

Abstract

This paper outlines the design of a code comparison tool, *ctcompare*, which use short sequences of lexical tokens from source code as a key in an inverted index to perform the code comparison. This technique allows the comparison of multiple source code trees simultaneously. Other significant features of the tool include the definition of a serialised token stream format which allows the independent analysis of a source tree without revealing the full source code, and isomorphic code comparison to identify renamed identifiers.

1 Introduction

There are various reasons to search for code similarity between separate trees[1] of source code, including (and not limited to) detection of proprietary code illegally used in another program, detection of open source code used in a proprietary program, detection of plagiarism by computer science students, determining the genealogical relationship between code trees separated by time (e.g. various versions of UNIX), and the identification of common code within various products written by a single company. In a similar vein, code clone detection identifies source code which is duplicated within a single source code tree; the aim here is identify duplicate code so that it can be eliminated.

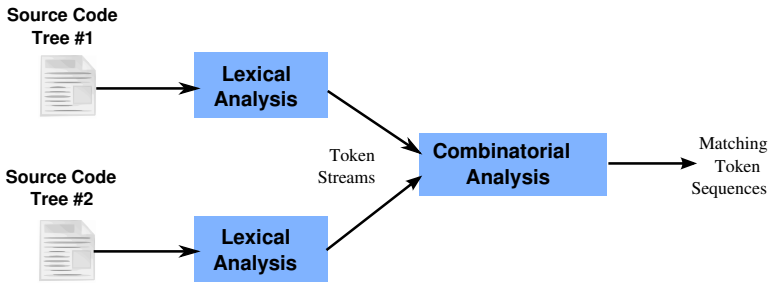
This paper outlines the development of a code similarity detection tool using a lexical approach: *ctcompare*. The advances made with this tool are:

- the definition of a serialised token stream format which allows the independent analysis of a source tree without revealing the full source code,
- the ability to compare multiple source code trees simultaneously,
- isomorphic code comparison, whereby the similarity between two code trees can be detected even when identifiers in one tree have been renamed in another tree.

2 Lexical Analysis

Code similarity detection using a lexical analysis approach is typically performed as follows. The source code from each tree undergoes lexical analysis to determine the individual lexemes or tokens in each tree. Using the token sequences from each tree, combinatorial analysis is performed to find the longest sequences of tokens common to both trees. These token sequences indicate code similarities.

[1] A tree of source code contains many individual files of source code.



3 A Serialised, Exportable Token Stream

In traditional lexical code similarity tools, the lexical and combinatorial analyses are performed at the same time by the same tool, so the stream of tokens from each source tree are internal to the tool. The tool thus requires that the actual source code to both trees be available for any code comparison to be performed.

This requirement for source code availability limits the use of such a tool. As an example, consider the situation where one company believes that its source code has been misappropriated and used as part of a competitor's product. Without access to the competitor's source code, the company is unable to determine the likelihood of such code theft.

In version 1 of the *ctcompare* tool, the decision was made to specify a file format for the token stream generated by the lexical analysis stage of the code comparison. Such a file format allows the lexical details of a source code tree to be exported in a serialised file form, without revealing the full details of the original source code.

The aim of having an exportable token stream in file format is to remove the limitation of requiring both source trees to be available in order to perform code comparison. Instead, one needs only the two token stream files in order to perform the comparison. In the above example of a company which believes that its source has been misappropriated, the company could ask a court for an independent witness or amicus curiae to perform a code similarity comparison. Both companies could provide the amicus curiae with an exported token stream, thus avoiding the need to reveal their original source code.

Any such token stream must therefore allow two conflicting goals to be achieved:

1. The stream must reveal enough of the original source code so that useful code similarity comparisons can be performed between trees: all similarities should be found, and few "false positives" should be reported.

2. The stream must not reveal enough of the original source code to allow that source code to be reverse engineered.

The design of the Code Token File (CTF) format used by the *ctcompare* tool balances both requirements. A CTF file represents the lexical elements of multiple source code files in a tree in a serialised form, with one variable-length record for each file in the tree. Each record begins with the source file's name and the time of its last modification. This is then followed by tokens which represent the structural elements of the source code. Most tokens are one octet in length and represent keywords, operators, brackets and parentheses. However, there are 5 specific token types with are represented by an octet for the token, and 2 octets for the token's value:

IDENTIFIER An identifier such as a variable, constant, or function name.

INTVAL An integer numeric literal.

CHARCONST A character literal.

STRINGLIT A string literal.

LABEL A label name.

The 2 octets following the token are used to store a 16-bit hash of the original textual value of the identifier, literal value or label. For example, consider the following short fragment of C code:

```
void print_word(char *str) {
  char *c;

  c=str;
  while ((c!='\0') && (c!=' '))
    putchar(c++);
  putchar('\n');
}
```

Keywords such as `void` and `while`, operators such as `*`, `=`, `!=` and `++`, and the parentheses and braces are represented as 1-octet tokens. The literal and identifier values are hashed down to 16 bits, and stored after their respective token octets. Thus, given the CTF representation of the above code, the following reconstruction would only be possible:

```
void id13043 ( char * id6928 ) {
  char * id23749;


  id23749 = id6928;
  while ((id23749 != 'c7388') && (id23749 != 'c20789'))
    id32886(id23749++);
  id32886 ('c12652');
}
```

The hash value for identifiers and literal values allows code trees to be compared, as the hash value from one tree can be compared to a hash value in a second code tree; at the same time, the hash does not reveal the original literal or identifier value, although a dictionary attack on the hash would provide a set of potential identifier or literal values.

A 16-bit hash value for identifiers and literal values was chosen to balance the effectiveness of the code comparison with the need to reveal too much of the original source code. A larger hash size (e.g. 32 bits) would eliminate false positives in the code comparison, but increase the likelihood of a successful dictionary attack (due to the decrease in hash collisions) and so reveal too much detail of the original source code. Similarly, a smaller hash size (e.g. 8-bits) would produce many false positives in the comparison, but minimise the effectiveness of a dictionary attack on the hash values.

4 Deficiencies of Version 1 of *ctcompare*

Version 1 of the *ctcompare* tool followed the traditional lexical/combinatorial analysis technique. The algorithm for the the combinatorial analysis stage was thus:

```
for each token position T1 in the first code tree {
  for each token position T2 in the second code tree {
    skip if the tokens at these positions (or their hash values) do not match;

    starting at T1 and T2, compare successive tokens in each tree
    (and any hash values), stopping on a mismatch or end of source file;

    skip if the length of the token run found is below a specified threshold;

    report the length of the token run found, and the token sequence;
  }
}
```

The tool's performance was effectively $O(M * M)$ where M and N are the number of tokens in each of the two source trees. An earlier paper [4] describes the attempts to improve the performance of the tool, including the use of Rabin-Karp [1] and a rolling hash across a fixed number of tokens. Still, the run-time performance of *ctcompare* version 1 was anything but ordinary.

4.1 Minimal Length for Similar Token Sequences

The above algorithm will find all sequences of tokens where there is a match between the two code trees. In practice, runs of matching tokens below a certain length can be discounted. The algorithm indicates that token sequences below a specified threshold would not be reported by *ctcompare* version 1, but what is a suitable threshold?

If, for example, only two consecutive tokens are considered a match, then there will be a large number of '`if (`' and '`while (`' matches found. The author considered a number of generic code fragments which would occur frequently in most independently-written C code; examples are shown in the table below.

Code Snippet	Purpose	Token Length
for (i=0; i<val; i++)	Standard FOR loop	13 tokens
if (x>max) max=x;	Find maximum	10 tokens
err= stat(file, &sb); if (err)	Open UNIX file	14 tokens

From this evaluation, the author chose a threshold of 16 tokens to be the minimum to indicate any significant form of code similarity; a token sequence common to two code trees which is below 16 tokens in length is not considered significant and is ignored. This is the main *a priori* design decision for *ctcompare* versions 1 and 2.

5 Efficient Comparison of Multiple Code Trees

One of the main purposes for *ctcompare* is to allow for the comparison of the various flavours and versions of the UNIX operating system, to determine genealogical relationships between versions, and to observe the changes in the source code over time and down different branches of the UNIX family tree.

Due to the large number of UNIX versions (see [2]) and the poor $O(M * M)$ performance of *ctcompare* version 1, the author posed the following question: is it possible to compare multiple code trees simultaneously, and can this be done efficiently? After much rumination on the problem, the 16-token similarity threshold prompted the following hypothesis:

> *Two code trees have a similarity if a token sequence of at least 16 tokens is common to both trees. Therefore, use a 16-token code sequence as a key to build an inverted index of multiple code trees which contain this sequence.*

In version 2 of *ctcompare*, a source code tree is tokenised and the token stream is exported as a CTF file. The token stream is then broken into 16-token sequences: for a stream of N tokens, there will be $N - 15$ token sequences; any identifier/literal hash values associated with the tokens in the sequence are removed. For the rest of the paper, a 16-token sequence will be called a token *tuple*.

Each tuple from the CTF file is used as a key to insert the following record into a database:

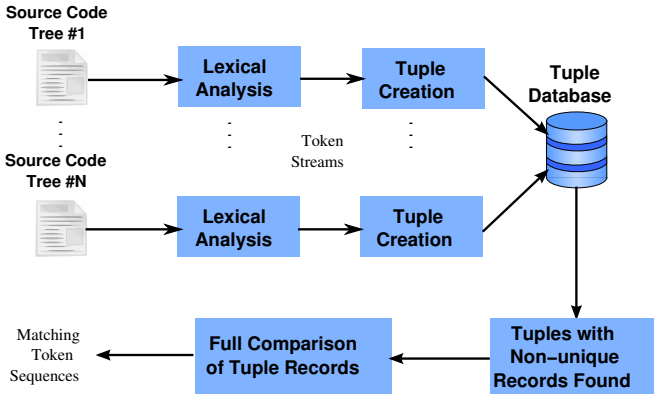
- which tree (of the multiple trees in the database) contains the tuple,
- which source file from the tree contains the tuple, and on what line number, and
- the offset in the CTF stream for the tree where this tuple occurs.

Each tuple is in effect an indicator of the "potential" for code similarity: two trees may share the tuple, but in fact the actual code similarity between the trees may be much larger than 16 tokens; this will be evaluated later.

After the insertion of "potential" token tuples from multiple source trees into the database, we have the following:

- some tuple keys have but a single database record: this indicates that the tuple is unique across all of the code trees, and does not represent any code similarity;
- some tuple keys have multiple database records: this indicates that the token sequence occurs in multiple files across all of the code trees, and may represent code similarity.

The structure and operation of *ctcompare* version 2 can now be outlined:



- multiple source code trees are tokenised into token streams and stored in CTF files;
- the token stream from each code tree is split into many 16-token tuples;
- the tuples and their location in the token stream are inserted into the database;
- with the database created, keys with multiple records are extracted from the database; each tuple key represents potential code similarity;
- all possible record pairs for a tuple are then compared to find the full extent of the code similarity beginning at the tuple;
- the full extent of the code similarity for each pair is output. As the minimum similarity is guaranteed to be 16 tokens or more, all matches meet the *a priori* 16 octet threshold.

6 The Comparison Algorithm

The tuple approach to code comparison allows the comparison of multiple trees, as for each tuple key in the database there can be records from multiple code trees. At the same time, for each tuple key there may be multiple records from the same code tree.

If we wish to perform code comparison between multiple trees, then we need to exclude the combinations of record pairs which are from the same tree. On the other hand, the tuple approach allows us to perform code clone comparisons, by including the combinations of record pairs which are from the same tree.

The basic algorithm for the comparison of tuples to find their full similarity is thus:

```
for each key in the database with multiple record nodes {
  obtain the set of record nodes from the database;

  for each combination of node pairs Na and Nb {
    if (performing a cross-tree comparison)
      skip this combination if Na and Nb in the same tree;
    if (performing a code clone comparison)
      skip this combination if Na and Nb in the same file;

    perform a full token comparison beginning at Na and Nb to determine
    the full extent of the code similarity, i.e. determine the
    actual number of tokens in common;

    add the matching token sequence to a list of "potential runs";
  }
}
```

```
walk the list of potential runs to merge overlapping runs, and
remove runs which are subsets of larger runs;

output the details of the actual matching token runs found.
```

7 Details of the Full Token Comparison

Each tuple from the database with multiple records indicates potential for code similarity of at least 16 tokens. However, the actual length of the code similarity may be larger; the records which the tuple points to must be compared pairwise to evaluate the full code similarity between two trees.

At the same time, two database records which share the same tuple may actually not share code similarity. Recall that the tuple key in the database contains 16 tokens, and no details of identifiers, literal values nor labels. This implies that the following two code snippets would create the same 16-token tuple:

```
if ( xx < yy ) { xx = 2 + yy * 7 ; }
if ( aa < bb ) { cc = 8 + dd * 5 ; }
```

Thus, the evaluation function must determine the full extent of code similarity, rejecting any resulting runs of similar tokens less than 16 tokens in length.

Given database records Na and Nb, the evaluation algorithm simply finds the start of each tuple in their respective token streams, and compares tokens (and token hash values where appropriate) until a mismatch occurs. The result is a "token run", stored in a structure which holds:

- details of the original tuples Na and Nb, which includes their start positions in the respective token streams,
- the length of the run in tokens,
- the end position of the run in each of the token streams, and
- the end line number of the run in each source tree, for ease of output later on.

7.1 Optimisation: Hashing the Identifiers in a Tuple

The previous section highlighted a deficiency in the tuple mechanism whereby some dissimilar code sequences would generate the same 16-token tuple. A small modification to the tuple concept helps to alleviate this shortcoming.

Along with the 16 tokens used to form a tuple, the database is modified to include a 16-bit hash of all the identifier/literal/label hash values which are associated with the tokens in the tuple. This now makes the tuple 18 octets in length. Two code snippets can now share an 18-octet tuple if and only if they have the same sequence of 16 tokens and they share the same hash of identifier hashes. This virtually eliminates the occurrence of dissimilar code sequences which share the same tuple.

8 Isomorphic Code Comparison

As seen above, the hash values for identifiers and literals can be used to detect if the identifiers in one source tree match with those used in another source tree. This comparison can be taken one step further to identify code which is similar, even when the identifiers from one tree have been renamed in another tree.

This is achieved by determining if two code fragments are *isomorphic*. Code which is isomorphic can be detected if there is a 1-to-1 relationship between identifiers. Take, for example, the following two code fragments.

```
int maxofthree(int x, int y, int z)
 {
   if ((x>y) && (x>z)) return(x);
   if (y>z) return(y);
   return(z);
 }

int bigtriple(int b, int a, int c)
 {
   if ((b>a) && (b>c)) return(b);
   if (a>c) return(a);
   return(c);
 }
```

The variable names have been changed, yet the algorithm is identical. If the order of occurrence of each identifier in each fragment is recorded, it can be determined if there is a 1-to-1 correspondence between them. Here, there is:

Identifier	Tag		Tag	Identifier
x	id1	$\Leftrightarrow$	id1	b
y	id2	$\Leftrightarrow$	id2	a
z	id3	$\Leftrightarrow$	id3	c

Note that there must be a 1-to-1 correspondence in both directions. If a new identifier `q` in the first fragment is introduced which appears to correspond to `b`, then this ends the similarity between the fragments as `b` already corresponds to `x`.

Both version 1 and 2 of ctcompare provide a run-time flag to enable isomorphic comparison, and a run-time limit to the number of entries in the isomorphic table. Consider the following function declarations:

```
int     ptsopen(),ptsclose(),ptsread(),ptswrite(),ptsstop();
int     ptcopen(),ptcclose(),ptcread(),ptcwrite(),ptcselect();
```

With an infinite sized isomorphic identifier table, any source code which declares 10 functions returning `int` would match the above two lines, regardless of the function names. *Ctcompare* sets a default isomorphic limit of 3 identifiers to limit the extent of this problem.

9 Efficiency in Merging the Token Runs

Tuples in the database are not guaranteed to be stored in any order, but a token run of full length $N > 16$ between two source trees is guaranteed to be represented in the database by $N - 16 + 1$ tuples, hence producing $N - 16 + 1$ runs: one of length 16, one of length 17, ... one of length $N - 1$ and one of length N. All of the token runs found by the evaluation function must be eliminated, except the longest.

Note that, for all of these overlapping runs, the following is true:

- all of the token runs end at the same tokens in each token stream, and
- the longest token run begins at the earliest position in the token streams.

We use this knowledge to efficiently merge the overlapping token runs. During the run evaluation phase, token runs produced by the evaluation function are stored in an AVL tree (known as the "potential" AVL tree), in the order of their earliest token position in the stream from one source tree.

During the merge phase, the potential AVL tree is traversed in key order: this ensures that the longest token runs are processed first. The position of the end token from the run is used as a key to search a second AVL tree: the "actual" AVL tree. This second tree contains the token runs found to be the longest, and is keyed by the position of the end token in the token run.

Consider a token run T of length 20, which has been selected from the potential AVL tree. If T is not the longest run in the set of overlapping token runs, then T's longest relative must have previously been selected during the processing of the potential AVL tree and stored in the actual AVL tree. Therefore, using T's end token position as a key, T's longest relative will be found in the actual AVL tree. T and T's longest relative can be compared, and T eliminated.

Alternatively, if T is the longest run in a set of overlapping token runs, then T's end token position will not yet appear in the actual AVL tree; therefore T will not be eliminated, and T can be inserted into the actual AVL tree for later comparison.

The overall algorithm is thus:

```
for each token run T in the potential AVL tree {
  search for a token run in the actual AVL tree which
  ends at the same token position;

  if (a token run A from the actual AVL tree is found with a
      matching end position AND T is a subset of A)
    discard T;
  else
    insert T into the actual AVL tree;
}
```

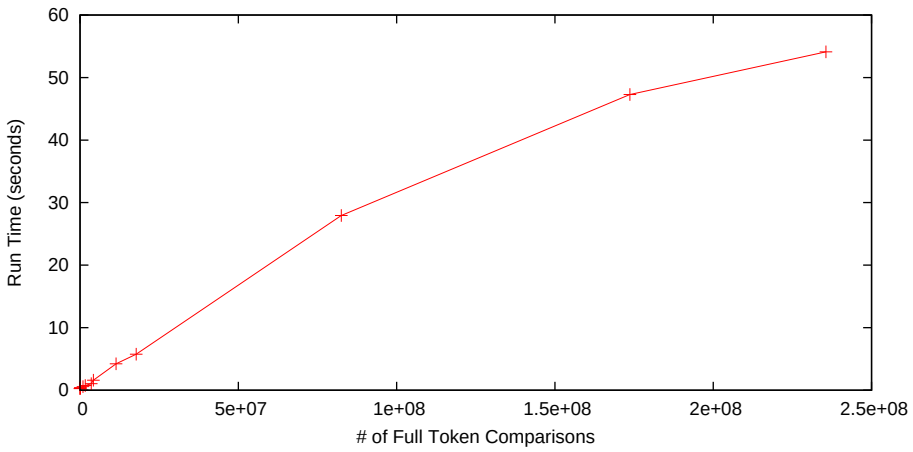
10 Overall Performance

Version 1 of the *ctcompare* tool could only compare two source trees at a time, and the similarity algorithm was order $O(M * N)$, where M and N were the lengths of each tree in terms of tokens.

Version 2 of the *ctcompare* tool uses several algorithms and heuristics. This makes it difficult to perform a theoretical analysis of the tool's performance, so an experimental analysis has been conducted. The experimental platform is a Dell Optiplex GX860 desktop PC with a 2GHz Pentium 4 CPU, 1 Gbytes of RAM and a 160Gbyte Western Digital hard drive.

To profile the performance of the tool, the source code of 14 code trees (many of which are flavours of UNIX and are thus related) are tokenised and successively added to the database of tuples. After each database insertion, version 2.5 of the *ctcompare* tool is executed to compare all code trees in the database; the wall-clock time for each comparison is recorded. The system is otherwise idle except for the code comparison. The 14 code trees together contain approximately 950,000 lines of non-blank C code.

Code profiling of *ctcompare*, using the *gprof* profiling tool, indicates that around 30% of the CPU time is taken up by the full token comparison of the 16-token tuples; no other function or algorithm takes such a significant percentage of the overall CPU time. A graph of the run-time versus the number of full token comparisons performed (shown below) appears to confirm that the performance of *ctcompare* is roughly linear with respect to the number of full token comparisons performed.



The use of 16-token tuples as an inverted index has increased the performance of *ctcompare* version 2 over version 1, but there have been a number of more prosaic changes which have improved the performance the tool. The use of Unix *mmap()* instead of the Unix Standard I/O (stdio) library to read the token sequences from the CTF files increased the performance eight times, and some hand coding of the full token comparison also doubled the tool's performance.

Ctcompare versions 1.3 and 2.5 were given the following pairs of source code trees. The table outlines the absolute run-times of both versions on the experimental platform described above.

Trees, Lines of Code	Version 1.3	Version 2.5
32V (10K), Net/2 (31K)	31s	0.22s
32V (10K), 4.4BSD (60K)	79s	0.37s
SysVR4 (180K), 4.4BSD (60K)	10221s	2.47s

On the same system, a code comparison of the 14 code trees representing 0.95M lines of non-blank C code takes 54.1 seconds of wall-clock time, resulting in 287,779 non-isomorphic code similarity results found (of at least 16 tokens) between the 14 code trees.

10.1 Code Clone Detection Performance

The *ctcompare* tool can be configured at compile-time to also perform code clone detection within a single code tree. To assess the tool's clone detection performance, a significant portion[2] of the FreeBSD 6.3 kernel source code was tokenised and inserted into a tuple database, totalling 2,600 source files and 1.2MLOC. *Ctcompare* found 602,549 instances of code clones in 177 seconds. The result reveals over 30 instances of code clones which are 500 tokens or more in length. Excluding header files, the top 10 source files sharing code in the FreebSD 6.3 kernel are:

Tokens Shared	File 1	File 2
865	kern/md4c.c	netncp/ncp_crypt.c
822	netipsec/key.c	netkey/key.c
775	arm/arm/busdma_machdep.c	i386/i386/busdma_machdep.c
756	boot/powerpc/loader/metadata.c	boot/sparc64/loader/metadata.c
728	boot/i386/libi386/comconsole.c	boot/pc98/libpc98/comconsole.c
717	boot/i386/libi386/vidconsole.c	boot/pc98/libpc98/vidconsole.c
663	nfs4client/nfs4_vfsops.c	nfsclient/nfs_vfsops.c
556	nfs4client/nfs4_vnops.c	nfsclient/nfs_vnops.c
538	i4b/layer1/isic/i4b_l1fsm.c	i4b/layer1/itjc/i4b_itjc_l1fsm.c
538	i4b/layer1/ifpnp/i4b_ifpnp_l1fsm.c	i4b/layer1/itjc/i4b_itjc_l1fsm.c

11 Verification of Correct Output

In the USL vs. BSDi court case in the 1990s, USL alleged the existence of significant amounts of UNIX 32V[3] code in the Net/2[4] distribution from the University of California, Berkeley, which had been released under a BSD license. Kirk McKusick [3] filed a deposition in the case in which he stated:

> Keith Bostic and I have reviewed Net/2 in detail since this lawsuit began (before the deposition was taken), spending hundreds of hours comparing it to 32V. Based on this exhaustive search I have found only 56 lines of [Net/2] code in five kernel files that appear to match lines of code in 32V. These 56 lines are out of the total of 539 source files and 230,995 lines of source code in the Net/2 kernel.

Ctcompare performs a non-isomorphic comparison of the two trees in 0.48 seconds. Of the 56 lines found by hand, *ctcompare* finds 26 lines of similarity. Some of these can be explained by code similarities of less than 16 tokens; others are identical comment lines which *ctcompare* discards.

[2] The selection was limited to ensure that ctcompare did not swap during its execution. The following directories were omitted: alpha/, amd64/, contrib/, dev/, ia64/, pc98/, powerpc/ and sparc64/.

[3] 32V was the first version of UNIX to run on the 32-bit VAX minicomputer.

[4] Net/2 was the second distribution of UNIX-free networking and kernel code from UCB.

In a private communication, McKusick reveals that "We were trying hard to find *anything* [in Net/2] that could be claimed to match [32V]". Indeed, for the Net/2 file `ufs/ufs_vnops.c`, the deposition states: "Being generous, 26 of the 1,863 lines of text in `ufs/ufs_vnops.c` match code in 32V". McKusick's original notes of the analysis differ: "Approximately 10 lines of source code in [`ufs/ufs_vnops.c`] have a strong [similarity] to the 32V versions of the routines".

Clearly, McKusick's original deposition overstates the amount of matching code between 32V and Net/2, and *ctcompare* finds nearly all but a few lines of actual similar C code. *Ctcompare's* isomorphic comparison does, however, find several other runs of similar code which were not found by McKusick, such as:

Net/2

```
if (bswlist.b_flags & B_WANTED) {
  bswlist.b_flags &= ~B_WANTED;
  thread_wakeup((int)&bswlist);
}
```

32V

```
if (bfreelist.b_flags&B_WANTED) {
  bfreelist.b_flags &= ~B_WANTED;
  wakeup((caddr_t)&bfreelist);
}
```

12 Limitations of Lexical Code Comparison

Lexical based code similarity comparison has several advantages: similarities can be detected despite structural rearrangement of code, the input trees do not have to be in a compilable state to be compared[5], and isomorphic comparison can detect code similarities even when identifiers have been renamed. However, the lexical approach does have at least one drawback.

In many languages, there are program elements which do not directly produce executable code. For example, in the C language, a source file may `#include` several other source files; structures of related variables may be declared, constants defined, and tables of literal values defined. The lexical approach might find definite code similarity between two source trees, and while the similarities are structurally significant, they may be semantically insignificant.

Examples from the comparison of 32V and Net/2 highlight this problem; each example below is headed by a line indicating the number of tokens in the token run, the details of the source files and corresponding line numbers; the header is followed by the actual code similarity from both files, separated by a dividing line.

```
31  32V/sys/h/nuser.h:96-106  Net2/sys/sys/errno.h:40-50
#define EPERM   1
#define ENOENT  2
#define ESRCH   3
#define EINTR   4
#define EIO     5
#define ENXIO   6
#define E2BIG   7
#define ENOEXEC 8
#define EBADF   9
#define ECHILD  10
#define EAGAIN  11
====================================
#define EPERM           1               /* Operation not permitted */
#define ENOENT          2               /* No such file or directory */
```

[5] This is important for C code, which can contain many #ifdef directives, and so can be compiled in multiple different ways.

```
#define ESRCH           3               /* No such process */
#define EINTR           4               /* Interrupted system call */
#define EIO             5               /* Input/output error */
#define ENXIO           6               /* Device not configured */
#define E2BIG           7               /* Argument list too long */
#define ENOEXEC         8               /* Exec format error */
#define EBADF           9               /* Bad file descriptor */
#define ECHILD          10              /* No child processes */
#define EDEADLK         11              /* Resource deadlock avoided */
```

The list of Unix error value names (EPERM, ENOENT etc.) is defined by the POSIX P1003.1 standard. While the numeric values are not defined by any specific standard, they are universally used by all versions of Unix, and the various Unix-like systems including Minix, Linux, the BSD family of operating systems. The general consensus is that the Unix error values are *scènes à faire* and not protectable by copyright.

```
17  32V/sys/sys/hp.c:64-68  Net2/sys/vax/uba/uda.c:2204-2208
        8800,   60,     /* cyl 60 thru 114 */
        0,      0,
        0,      0,
        0,      0,
        0,      0,
====================================
        -1,     0,      /* C=blk 0 thru end */
        0,      0,
        0,      0,
        0,      0,
        0,      0,
```

This second example shows a sequence of literal values, used independently in two unrelated tables, is detected by a lexical comparison mechanism as being similar code.

```
16  32V/sys/sys/cons.c:89-92  Net2/sys/i386/isa/wd.c:897-900
consioctl(dev,cmd,addr,flag)
dev_t dev;
caddr_t addr;
{
==================================
wdioctl(dev,cmd,addr,flag)
        dev_t dev;
        caddr_t addr;
{
```

In the Unix family of operating systems, direct access to devices is performed via the *ioctl()* system call. In the above example, two different *ioctl()* interfaces to two different hardware devices in two different code trees use the same function prototype; again, this is lexically identical, but the code is semantically unrelated.

13 Future Work and Acknowledgments

The prime area of CPU usage in *ctcompare* is the full token comparison and evaluation of a token run, given a tuple and a pair of database records. If the tuple keys in the database could be traversed in order of starting token position, this would allow *ctcompare* to find the longest token runs first; subsequent tuples which are subsets of a longer run could be identified before a full token comparison is performed

on the tuple and associated database records. This would require some significant modifications to the *ctcompare* tool, but may improve the tool's performance further still.

At present, *ctcompare* and the associated tools to tokenise and manipulate the tuple database are written as Unix command-line utilities. Comments supplied by several users of ctcompare indicate a desire that *ctcompare* be developed into a Unix shared library with a well-defined API, so that the code comparison algorithm can be embedded into other programs.

The author would like to thank Michael Stefaniuc from RedHat Inc. for the code to replace Standard I/O with *mmap()* and for his other suggestions; the author also thanks Paddy Krishnan, Phil Stocks & Marcus Randall at Bond University for their suggestions, encouragement and for reviewing drafts of this paper.

References

[1] D. Ellard. The Rabin-Karp Algorithm, Jul 1997. http://www.eecs.harvard.edu/~ellard/Q-97/HTML/root/node43.html.

[2] É. Lévénez. UNIX History, Apr 2008. http://www.levenez.com/unix/.

[3] M. K. McKusick. Second Declaration of Dr. Kirk McKusick in Support of the Regents of the University of California's Amicus Brief Re Motion for Preliminary Injunction, Jan 1993. http://minnie.tuhs.org/UnixTree/Net2Kern/930119.mckusick.decl.2.

[4] W. Toomey. Comparing C Code Trees. In *Proceedings of the 2004 AUUG Winter Conference*, pages 55–61, Sep 2004.