Code Similarity Detection in Multiple Large Source Trees using Token Hashes

Dr. Warren Toomey

School of IT,
Bond University
Queensland. Australia

Abstract

The ability to find similarities between two source code bases, or within one code base, has many uses including the detection of student plagiarism, the identification of intellectual property violations and the location of repeated code in a code base amenable to refactoring. Previous structure-metric approaches have used either suffix trees or modified Longest Common Subsequence algorithms to detect code similarities. This paper introduces *ctcompare*, a tool which uses hashes on fixed-length token sequences to find code similarities. One significant benefit of this approach is the ability to search multiple large trees of source code simultaneously for similarities. A performance analysis of *ctcompare* on a large body of code is also given.

Keywords: Code similarity, code clone, clone detection, code redundancy, plagiarism, software.

1 Introduction

The issue of source code similarity is of interest to many areas of the computing milieu. In an academic environment, code similarity between two or more student submissions may indicate plagiarism or collusion between the students. In the software industry, code similarity between products created by two different organisations may indicate theft of source code and other intellectual property. Code similarity in different sections of a single product may highlight the need for the product's source code to be refactored to remove any code duplication. And in the new field of computing archaeology (Hunt & Thomas 2002), the detection of code similarity can identify genealogical relationships between software systems separated by time and by organisation, such as the multitudinous family members of the Unix operating system (Lévénez 2008).

There are several approaches to the detection of code similarity between two or more code bases. In a *textual comparison* approach, lines of source code are compared textually, and a similarity metric given based on the number of lines in common (Ducasse, Rieger & Demeyer 1999). In a *counting-metric* approach, features such as the number of identifiers, keywords and syntax elements are counted on fragments from each code base to produce a set of feature vectors (Mayrand, Leblanc & Merlo 1996); the distance between these vectors are then compared to find possible code similarity. By parsing each code base and creating *abstract syntax trees*, subtrees are compared to search for code similarity (Baxter, Yahin, Moura, Sant'Anna & Bier 1998). And in the *concept clone* approach, features such as abstract data types and algorithms are identified in each code base and compared for similarity (Marcus & Maletic 2001). Several papers provide an excellent overview of these different approaches (Bellon, Koschke, Antoniol, Krinke & Merlo 2007, Roy, Cordy & Koschke 2009).

This paper concentrates on one other approach to the detection of code similarity, the *structure-metric* approach which uses lexical analysis. Each code base to be compared is first lexically analysed to produce a sequence of tokens. These token sequences are then compared to find common token subsequences which indicate similarities between the code bases. We outline a new approach to the structure-metric approach which uses hashes of token tuples. The benefits of this approach over the existing research include the simultaneous comparison of multiple large code bases, fast performance and the export of serialised token streams for sensitive (e.g. proprietary) code bases.

2 Previous Work

Baker's approach to code comparison is to create token sequences from each input line of code (Baker 1995). Each sequence is passed through a function which, after separating out the literals and identifiers, generates a value for the sequence. The literals and identifiers are encoded as a parameter to the function; the encoding preserves the order of literals and identifiers, so code similarity can be detected even when literals and identifiers are renamed. Two lines of code are deemed similar if their function value plus encoded parameters match. The function values plus encoded parameters are stored in a suffix tree (Gusfield 1997) which can be built and searched in time linear to the input size, although Baker notes that some worst-case inputs lead to quadratic running time. The line-based approach does fail to identify code similarity when lines of code have either been split into multiple lines or multiple lines merged together. Kamiya *et al.* also use token sequences and a suffix tree to find *code clones*: duplicated code within a single large code base (Kamiya, Kusumoto & Inoue 2002). While not line-based, token sequences are created that begin with certain tokens (e.g. '{', selection keywords such as 'if', 'for' & 'while', declaration keywords such as 'class', 'enum' & 'typedef' etc.). Time and space complexity is $O(m.n)$, where m is the length of the largest code clone and n is the input length in tokens.

Token sequences can be viewed as strings, and so detecting code similarity is a form of finding the Longest Common Subsequence (LCS) between two strings (Bergroth, Hakonen & Raita 2000). Several LCS algorithms have been used in code comparison. Traditional LCS algorithms such as the Levenshtein distance algorithm (Levenshtein 1966) preserve substring ordering, so if a section of source code in one input is moved, the algorithms identify this as a number of differences instead of a single change in the code. Alternatives have been proposed by Heckel (Heckel 1978) and Tichy (Tichy 1984) to overcome this limitation.

Both Wise (Wise 1996) and Prechelt *et al.* (Prechelt, Malpohl & Philippsen 2000) use the Karp-Rabin Greedy-String-Tiling algorithm (Wise 1993), a form of LCS which compares pairs of strings and also permits the transposition of substrings. The GST algorithm has two phases. In the first, a triply-nested loop searches for all tokens in code base A which match tokens in code base B, with the innermost loop attempting to extend the match to find the longest sequence of token matches. In the second phase, all the tokens that form sequences of tokens matches from phase 1 are marked so they will not be used for further matches in phase 1. Both phases are repeated until no further matches are found. GST has a time complexity of $O(N^3)$; both Wise and Prechelt then use the Karp-Rabin algorithm and other optimisations to reduce this complexity further to $O(N^{1.12})$ and $O(N)$ in practice, respectively.

3 Ctcompare: Code Comparison using Token Hashes

The aim of *ctcompare*, our tool to detect code similarity, is to take two or more large trees (i.e. multiple files) of source code as input, convert these into streams of tokens, and to find all sequences of tokens of length N or greater which occur in the code trees; the value for N is chosen at run-time. Our work draws upon, combines and extends the ideas of the previous research. As with Baker, we abstract a short sequence of tokens into a unique value: here, the sequence size is fixed at N tokens and a hash function produces the unique value.

These hash values are used to find code similarities of length N, but this is unlikely to be the maximal length of a code similarity. Existing LCS algorithms could be used here to find the full extent of the token sequence match; we have implemented an alternative algorithm which uses the token hashes, not the tokens themselves, to perform the token sequence matching. As a by-product of the approach, we can compare token sequences from multiple code trees at the same time whereas traditional LCS algorithms are only able to compare two strings simultaneously.

4 A Serialised, Exportable Token Stream

In other token-based structure-metric code similarity tools, the lexical and similarity analyses are performed at the same time by the same tool, so the stream of tokens from each source tree are kept internal to the tool. The tool thus requires 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 organisation believes 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.

With *ctcompare*, we decided 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 format without revealing the full details of the original source code. Instead, one needs only the resulting token stream files in order to perform the comparison. In the above example of an organisation that believes its source has been misappropriated, the organisation could ask a court for an independent witness or amicus curiae to perform a code similarity comparison. Both organisations 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 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 a sequence of tokens which represent the structural elements of the source code. Most tokens are one octet in length and represent keywords, operators and other syntax elements. However, there are 5 specific token types which are represented by an octet for the token followed by two octets for the token's value:

- an identifier such as a variable, constant, or function name;
- a numeric literal;
- a character literal;
- a string literal; and
- a label name.

The two 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, only the following reconstruction would 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. 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.

We chose a 16-bit hash value for identifiers and literal values to balance the effectiveness of the code comparison with the need to minimise full disclosure 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. A smaller hash size (e.g. 8-bits) would produce many false positives in the code comparison but minimise the effectiveness of a dictionary attack on the hash values.

5 Token Tuples, Tuple Description Nodes and the Run List

The aim of the *ctcompare* tool is to find all sequences of tokens of length N or greater which occur in two or more source code trees. The value of N is arbitrary, and the default run-time value of 16 tokens was chosen to exclude common short sequences of source code such as
`for (i=0; i < CONST; i++)`.

The comparison algorithm uses *tuples* (i.e. subsequences) of N consecutive tokens, along with any associated identifier/literal hash values, as indicators of code similarity. If multiple code trees share a common tuple, then this indicates code similarity of at least N tokens between the trees. The token tuples are also used to find the longest sequence of tokens shared between the code trees.

For each tuple of N tokens, *ctcompare* keeps a tuple description node (TDN) with this information:

- the identity of the CTF file from where this tuple originates;
- the offset within the CTF file of the first token in the tuple;
- the name of the source code file and the line number where the first tuple occurs; and
- a pointer to the TDN of the immediately preceding tuple, if any.

Tuples and the TDNs from all of the source code trees are kept by *ctcompare* in a data store, to be discussed.

Ctcompare reports the details of each specific run of similarity it finds: which source files are involved, the beginning and the end of the similarity. A run of code similarity is stored in a Run structure which contains:

- pointers to the TDNs in two code trees where the run of similarity starts;
- pointers to the TDNs in two code trees where the run of similarity ends; and
- the length of the run of similarity in tokens.

6 The Comparison Algorithm

The algorithm used by *ctcompare* to find code similarities in multiple code trees is shown as Algorithm 1 on the next page. Token tuples are used to find initial runs of commonality between code trees; the tuples are also used to extend these runs until the longest match is found.

The data store of tuples and associated TDNs is initially empty and grows as each new tuple is processed. This is done to prevent duplicate matches: for example, if all tuples are initially placed in the data store and if a search on tuple T_1 will find T_2, then a search on T_2 would also find T_1. This on-the-fly tuple insertion strategy is used in the current version of *ctcompare* which builds and keeps the data store in memory. The alternative approach of initially storing all the tuples in the data store would be required where the data store is persistent (e.g. stored on disk), but duplicate tuple matches would have to be dealt with. We used the on-disk approach in earlier versions of *ctcompare*, but we now choose to keep the data store in memory to optimise performance.

7 Implementation of the Algorithm

The above algorithm as given does not define a specific implementation; however, its efficiency depends on:

1. the cost of searching for a tuple and associated TDN in the data store;
2. the cost of inserting a tuple and and associated TDN into the data store;
3. the cost of extending an existing similarity run, given matching tuples T and T_2; and
4. the cost of adding a new similarity run to the list of similarity runs.

Search and insert operations on the data store need to have low complexity, as do the operations on the list of similarity runs. As will be shown in the next section, the TDN operations on the data store have the highest cost on the algorithm's performance.

The *ctcompare* tool as implemented keeps the data store and the list of similarity runs in memory. This also places a bound on the size of the data store and similarity list and hence the size of the input to the tool. During the implementation of *ctcompare*, we evaluated techniques such as binary search trees, AVL trees, single-level and multi-level hash tables. In the end we chose a single-level hash table with chains as this provided the best absolute performance and also the lowest memory usage.

Each tuple of N tokens and associated literal/identifier hashes is itself hashed to a 32-bit value. To access the data store, 24 bits of the hash value are used to find the bucket in a single-level hash table, and the bucket points to a chain of TDNs. Each TDN is augmented to hold the remaining 8 bits of the tuple hash so hash collisions at the bucket level can be resolved.

Inserting a new tuple and TDN into the data store is $O(1)$ as the hash function used is $O(1)$ and the TDN can be inserted at the beginning of the chain. Searching for matching tuples in the TDN is $O(N)$ where N is the size of the chain which may contain the matches.

7.1 Drawbacks of Using Tuple Hashes

While hashed tuples provides low complexity and high absolute performance, their use in the *ctcompare* algorithm also bring some drawbacks which must be addressed.

For a run of similarity of N tokens to be found between two code trees, the two tuples of size N must match. With the use of hashed tuples, there is now the likelihood of false matches due to the probability of hash collisions: $1/2^{32}$ for the 32-bit hash function. To avoid these false matches, the *ctcompare* algorithm is modified as follows:

- tuples of size $N-1$ (and associated TDNs and hash values) are created instead of size N; and
- the algorithm performs searches using $N-1$ sized tuples, but only similarity runs of size N are reported.

While the probability of two unrelated tuple hashes matching is $1/2^{32}$, the probability of the following two unrelated tuple hashes also matching is $1/2^{32} * 1/2^{32} = 1/2^{64}$. This eliminates false tuple matches for all practical purposes, albeit at a small performance cost.

A 24-bit hash table is also used (when two tuples match) to search the list of similarity runs for an existing run to extend. In this instance, the buckets of the hash table do not contain a chain of runs; rather, each bucket holds exactly zero or one similarity runs. When the hash table is searched, there is no possibility that the wrong run will be extended due to a hash collision, as each Run

Algorithm 1 *ctcompare*'s comparison algorithm

```
for (all tokenised source trees) {
  for (all consecutive sequences of N tokens from the source files in the tree) {
    build a token tuple T of the N tokens plus their identifiers, and build its TDN;
    for (each existing tuple T2 in the data store which matches T) {
      if (T and T2 would extend an existing similarity run R) {
        modify R so T and T2 are now the end tuples of the run;
      } else {
        create a new similarity run R where T and T2
                                are the start and the end tuples of the run;
        add R to the set of similarity runs;
      }
    }
    add tuple T and its TDN to the data store;
  }
}
output all of the similarity runs found;
```

node contains enough information to prevent this. However, when a new run needs to be inserted into the hash table, there is the chance that the bucket holds an existing Run node. This node is evicted from the hash table. This causes some runs of code similarity found to be reported as several smaller runs of similarity: no similarities will be lost, but their size may be misreported by *ctcompare*.

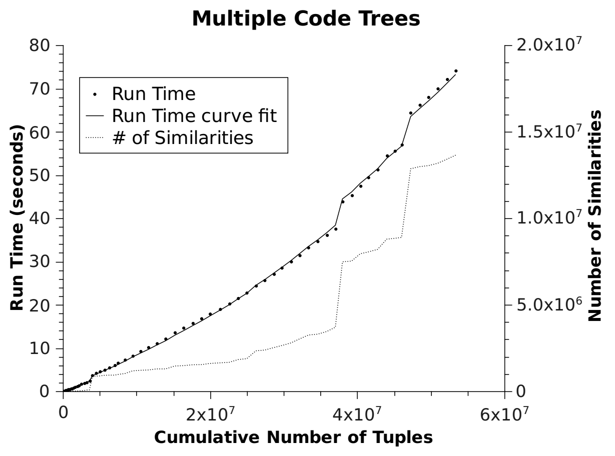
8 Performance Analysis

We measured the performance of the *ctcompare* tool using subsets of two data sets. The platform used was one core of a 3GHz Intel i5 CPU in a desktop PC with 4G of DDR3 RAM. The main limitation of the current version is that the tuple/TDN data store and the list of similarity runs must fit into main memory; therefore, we gave increasingly large data sets to *ctcompare* until main memory was exhausted.

We designed *ctcompare* to be able to find code similarities across multiple source code trees totaling millions of lines of code; thus, we chose a real-world data set to analyse the performance. This consists of several related Unix source code trees of various sizes along with some arbitrarily chosen code trees which have no similarities[1]. The code trees were arranged in order of increasing token count. Large code trees were split into separate inputs of approximately 2.5 millions tokens to increase the number of data points. Successively larger groups of code trees were given as inputs to *ctcompare* so we could measure the run time of the tool against the input size. We took measurements of the run time, the cumulative number of token tuples, and the number of runs of similarity found; the results are shown in the following graph.

In the graph, the x-axis represents the number of tuples in the input. The run time is shown as a scatter plot of points with a fitted curve, and the number of similarities found between the code trees is shown as a dashed line. The top-right point shows *ctcompare* can find 14.5 million runs of similarity from 13.2 millions lines of code across 35 code trees in 74 seconds.

[1]The data sets are available at http://minnie.tuhs.org/Programs/Ctcompare.



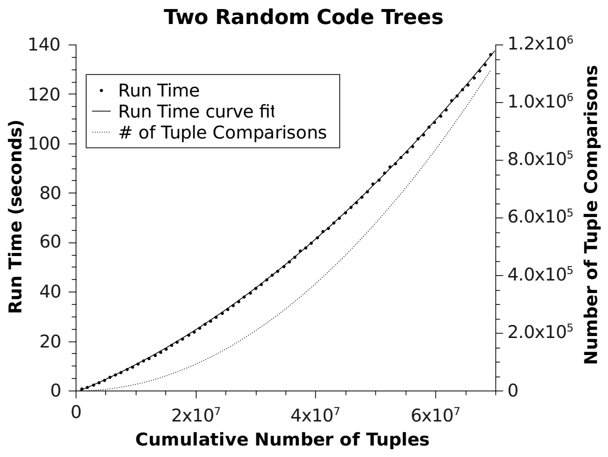
The scatter plot indicates the run-time complexity of *ctcompare* is $O(N^2)$ with respect to the input size in tuples (hence, tokens), as shown by the fitted curve[2]. The other factor in the tool's performance is number of runs of similarity found between the input code trees; this makes sense as the inner loop of the algorithm will depend upon the number of similarities. However, the $O(N^2)$ performance based on the input size is puzzling as the algorithm has only a single loop based on the number of tuples.

To isolate the quadratic behaviour we performed a second performance analysis on two large pseudo-randomly generated token streams, chosen so as to minimise any runs of similarity. The two streams were truncated at increasingly larger sizes so the run-time of *ctcompare* could be measured against an increasingly larger input size. The results are shown in the following graph.

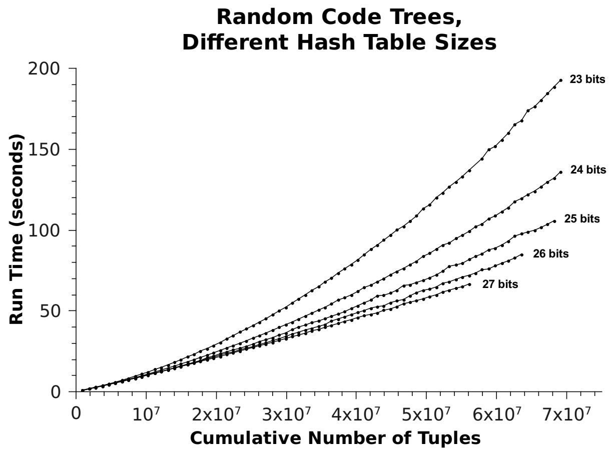
In the graph, the x-axis represents the number of tokens used as input. The run time is shown as a scatter plot of points with a fitted curve, and the number of tuple comparisons is shown as a dashed line. Only 17 runs of similarity are found for the top-right data point, so the run-time effect from the number of runs of similarity is negligible and can be discounted.

With two random code trees, *ctcompare* still exhibits an $O(N^2)$ run-time complexity based on the input size, and the number of comparisons between tuples from each tree is also $O(N^2)$. Given there were next to no similarities found between the two random trees, there should have been no matches between tuples in the two code trees. The number of tuple comparisons should have been close to zero and not 10^5 or more as shown.

[2]$runtime = 8.1 * 10^{-15} * tuplecount^2 + 6.1 * 10^{-7} * tuplecount + 1.3 * 10^{-6} * similarities$



This behaviour is due to the size of the index into the single-level hash table to find chained TDNs: if it is too small for the entire input, then the chains in each bucket become too long and hold many TDNs that do not match the one being searched for. By resizing the index into the single-level TDN hash table, as shown in the following graph, we can improve *ctcompare*'s performance and flatten its quadratic behaviour. This does come at the expense of main memory usage.



9 Other Features

One of *ctcompare*'s features is its ability to find code similarities across multiple code trees simultaneously. The algorithm will also find code similarities between files within one code tree. As token tuples for one tree are added to the data store, the line

```
for (each existing tuple T2 in the data
                     store which matches T) {
```

implies new tuples from a tree will be compared against previous tuples from the same tree. This allows *ctcompare* to find duplicated code within a single tree that is amenable to refactoring. To ensure *ctcompare* only finds similar code between trees, a run-time option selects between the above loop and this one:

```
for (each existing tuple T2 which matches T
                but not from the same tree ) {
```

Apart from the run-time choice of the value for N (the minimum number of tokens to consider for similarity) and the choice of comparison within or between trees, *ctcompare* also offers these options:

- output of the matching token streams, or output of the actual lines of code found to be similar; and
- to match identifiers exactly, or to detect remappings of identifiers.

Remapped identifiers occur for several reasons: students often rename variables to disguise plagiarism; during refactoring, names of variables or constants may be changed to make the code more readable. We search for remapped identifiers in a very different way to Baker.

If identifiers are renamed, then their 16-bit hash values will change and so we cannot include these hash values in the token tuple of size N. When searching for code similarities with remapped identifiers, we only hash the tokens in the tuple and leave out the identifier hash values. Once a potential similarity run is found and the algorithm attempts to extend the run, we keep a 1-to-1 correspondence table for the (hashed) identifiers found in the run, similar to the following:

Identifier	**Tag**		**Tag**	**Identifier**
x	id1	⇔	id1	b
y	id2	⇔	id2	a
z	id3	⇔	id3	c

At run-time, the maximum size of the table is set, e.g. 3 corresponding identifiers. The algorithm allows a run of similarity to be extended until either more identifiers are found than can fit into the table (e.g. a variable named 'q'), or if a correspondence between identifiers in the run violates the correspondence in the table (e.g. variable 'c' starts to match against variable 'y'). The run-time selection to search for remapped identifiers does impose a significant performance penalty on *ctcompare*.

Ctcompare uses single-octet tokens to create N-sized token tuples, so any mechanism which analyses an input stream into single-octet tokens can be used; in other words, *ctcompare* is source language neutral. We have written lexical analysers in *ctcompare* for these inputs: C and C++, Java, Python, Perl, assembly language, hexadecimal-encoded binary and plain text. The last analyser treats every word as a string literal; the comparison is therefore one of matching hashed sequences of hashed literals.

10 Future Work

Ctcompare is at present a mature tool and we cannot see any significant future changes to the algorithm for single-CPU use. There are still a few deficiencies that need to be addressed. The input size to *ctcompare* is constrained by the size of available memory. At present we have written *ctcompare* to run on 32-bit Unix and Linux systems; we would like to re-target the tool to work on 64-bit platforms. Not only would this allow for more than 4Gbytes of main memory, but it would also allow us to increase the size of the single-level TDN hash table. This would improve performance by reducing the size of the TDN chains in each hash bucket.

One other future option would be the modification of the algorithm to work across multiple CPUs or CPU cores, but this would require a significant reworking of *ctcompare*'s tuple insertion strategy along with associated data structure locking.

11 Conclusion

The token-based structure-metric approach to detect code similarity across several code trees has seen substantial research in the past. With our tool *ctcompare* we take a novel approach which deviates from the traditional use of suffix trees and Longest Common Subsequence variations. Instead, we take hashes of fixed-length token subsequences (along with hashes on identifiers and literals) and use these to find minimal runs of similarity. The token hashes are then used instead of the tokens to find the maximal length of the runs of code similarity. This also has the effect of eliminating any false positives due to initial hash collisions.

By using a single-level hash table with chains we have optimised the absolute performance of *ctcompare*, but the algorithm's relative performance is still $O(N^2)$ where N is the input size in tokens. This complexity can be flattened somewhat by increasing the hash size used as the index into the TDN hash table.

The algorithm used in *ctcompare* allows us to find code similarity within one code tree, two code trees or multiple code trees simultaneously. The use of hashes gives a compact representation of the input code, allowing us to compare tens of millions of lines of code within a 32-bit PC's available memory. Other features of *ctcompare* include the exporting of code trees in serialised token format and the ability to find remapped literals during the code comparison.

References

Baker, B. S. (1995), On Finding Duplication And Near-Duplication in Large Software Systems, *in* 'WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering', IEEE Computer Society, Washington, DC, USA, p. 86.

Baxter, I., Yahin, A., Moura, L., Sant'Anna, M. & Bier, L. (1998), 'Clone detection using abstract syntax trees', *IEEE International Conference on Software Maintenance* **0**, 368.

Bellon, S., Koschke, R., Antoniol, G., Krinke, J. & Merlo, E. (2007), 'Comparison and evaluation of clone detection tools', *IEEE Trans. Softw. Eng.* **33**, 577–591.

Bergroth, L., Hakonen, H. & Raita, T. (2000), 'A survey of longest common subsequence algorithms', *International Symposium on String Processing and Information Retrieval* **0**, 39.

Ducasse, S., Rieger, M. & Demeyer, S. (1999), 'A language independent approach for detecting duplicated code', *Proceedings of the IEEE International Conference on Software Maintenance* p. 109.

Gusfield, D. (1997), *Algorithms on Strings, Trees and Sequences*, Cambridge University Press.

Heckel, P. (1978), 'A technique for isolating differences between files', *Commun. ACM* **21**, 264–268. *http://doi.acm.org/10.1145/359460.359467

Hunt, A. & Thomas, D. (2002), 'Software archaeology', *IEEE Software* **19**, 20–22.

Kamiya, T., Kusumoto, S. & Inoue, K. (2002), 'CCFinder: a Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code', *IEEE Trans. Softw. Eng.* **28**(7), 654–670.

Lévénez, E. (2008), 'Unix History Timeline'. *http://www.levenez.com/unix/

Levenshtein, V. I. (1966), 'Binary codes capable of correcting deletions, insertions, and reversals', *Soviet Physics Doklady* **10**(8), 707–710. *http://sascha.geekheim.de/wp-content/uploads/2006/04/levenshtein.pdf

Marcus, A. & Maletic, J. (2001), 'Identification of high-level concept clones in source code', *International Conference on Automated Software Engineering* **0**, 107.

Mayrand, J., Leblanc, C. & Merlo, E. (1996), 'Experiment on the automatic detection of function clones in a software system using metrics', *Proceedings of the IEEE International Conference on Software Maintenance* pp. 244–253.

Prechelt, L., Malpohl, G. & Philippsen, M. (2000), 'Finding Plagiarisms Among a Set of Programs with JPlag', *Journal of Universal Computer Science* **8**, 1016–1038.

Roy, C. K., Cordy, J. R. & Koschke, R. (2009), 'Comparison and evaluation of code clone detection techniques and tools: A qualitative approach', *Science of Computer Programming* **74**(7), 470 – 495. Special Issue on Program Comprehension (ICPC 2008).

Tichy, W. F. (1984), 'The string-to-string correction problem with block moves', *ACM Trans. Comput. Syst.* **2**, 309–321. *http://doi.acm.org/10.1145/357401.357404

Wise, M. (1993), *Running Karp-Rabin matching and greedy string tiling*, Basser Dept. of Computer Science, University of Sydney.

Wise, M. J. (1996), 'YAP3: Improved Detection of Similarities in Computer Program and Other Texts', *SIGCSE Bull.* **28**(1), 130–134.