**Normalised Compression Distance for Clone Detection**

The Normalised Compression Distance (NCD) is a method that can be used to measure similarity between two given data objects, such data object could be files or search terms. The two selected data objects can be represented as data vectors, where these vectors are subjected to a compression algorithm. The compression algorithm computes a compressed length of each data vector and both data vectors concatenated. For example one might utilise Gzip as a compression method for the given data objects. The higher the compression value of the two concatenated data vectors relative to how each compressed data vector indicates that the more similar they are. The NCD between two data objects is captured more formally as:

Where is the compression method used. It was demonstrated that the NCD methods is generally applicable, parameter free, noise resistant and demonstrated theoretically to be optimal [1]. Furthermore NCD can be used with machine learning to induce object classification, where the classification can be achieved by utilising the computed NCD values. Such classification using NCD was shown to be effective by Stefan Axelsson [1]. Therefore it could be possible to apply this technique for clone detection in source code as the process of clone detection focuses on fragments within a source file that are similar.

Clone detection in source code heavily relies on a solid definition of what a clone is however, a single definition of what a clone is has yet to be universally agreed upon and thus clones are categorised with types. The clone types are as follows:

* Type 1 clones have identical code segments except for layout and comments.
* Type 2 clones are structurally identically with differences in identifier names, layout and comments.
* Type 3 clones are similar to type 2 but with variation in the addition, removal and modification of statements.
* Type 4 clones focus on the semantically equivalence of source files.

By using machine learning techniques it could be possible to have a training set that contains files that are labelled with the above clone types. For each one of the clone types it is then possible to calculate the NCD values and apply classification algorithms to partition the training set according to clone type. After the training phase, the new pairs of source file could then undergo similar procedures such that they could be classified as one of the clone types or none. As this technique would only examine the information distance between the source files, it would be difficult to find at a granular i.e. the source level, where the duplication/clone actually occurs. However, at a higher level it would be recommended to apply such technique.

**References**

1. USING NORMALIZED COMPRESSION DISTANCE FOR CLASSIFYING FILE FRAGMENT, Stefan Axelsson, Blekinge Institute of Technology, 2010 International Conference on Availability, Reliability and Security.

**Considerations for CCFinder as a Clone Detection Tool**

CCFinder is a tool that finds a code portion that is similar to another code portion in another source file with a basic idea of transforming a token sequence in a regularized form on which two code portions likely to be cloned are identical sequences of tokens. This tool is able to extract code clones in C, C++, Java and COBOL and any other source files.

CCFinder was designed based on four underlying concepts which are summarized below:

1. Tool should be robust such that it is applicable to large systems within affordable computation time and memory.
2. Tool should lay more emphasis on detecting clones that will help users using some heuristic knowledge. This helps improve effectiveness of clone analysis.
3. Tool should always report edited pasted codes or renamed variables
4. Tool should be independent of a language.

CCFinder and Clone detection has the following notable terms:

* **Clone Relation:** When two code portions have the same sequences, then we say a clone relation exist between the two code portions. i.e in order words equivalence relations on code portions
* **Clone Pair:** A pair of a code portion for a given clone relation.
* **Clone Class:** maximal set of code portions in which clone relation holds between any pair of code portions.

Being a token-based clone detection technique, the following steps are taken by the tool for measuring source code similarity:

1. **Lexical Analysis**: where lines of source files are divided into tokens corresponding to a lexical rule of the programming language. This stage also entails removing white spaces and comments
2. **Transformation**: The token sequences are transformed based on defined transformation rules and then each identifier are replaced with a special token making code portions with different variable names clone pairs.
3. **Match Detection**: From the substrings of the transformed token sequence, equivalent pairs are detected as cloned pairs.
4. **Formatting:** Each location of clone pairs is converted into line numbers on the original source files.

CCFinder is able to address the following clone detection issues:

* CCFinder is able to automatically identify and separate each function definition and each table definition code.
* CCFinders also transform complex namespace into simple form.

**Empirical Results:**

In performance analysis, CCFinder shows the possibility of achieving the following feats:

1. Working very well with C, C++, Java and COBOL source files.
2. Handling millions of line of Codes on three operating systems - FreeBSD 4.0, Linux 2.40 and NetBSD 1.5.
3. Evaluation of clones in terms of coverage (% LOC); percentage of lines that included any portion of clone and coverage (% files); percentage of files that included any clones.

However, it is reported that large number of clones was found between FreeBSD and NetBSD whereas few clones were discovered between Linux and the other two.

CCFinder uses the *“Suffix-tree matching”* algorithm which computes all same subsequence in a sequence composed of fixed alphabet that enables it to detect some clones that other tools that use the *“line-by-line matching*” algorithm would not detect. However, the computing complexity of the former is high and expensive, but more than 23% of the clones have always been detected with the *suffix-tree matching* technique when compared with *line-by-line* *matching.*

**Source code similarity measure using Plagiarism Detection Technique as in JPlag: Summary of Considerations**

**About the technique**

The Plagiarism detection approach for measuring code similarity applies the process of locating instances of plagiarism within a source code. Various algorithms are available for this approach and are classified based on the following:

Ability to match string segments; Ability to convert the program into tokens; Ability to build and compare parse trees; Ability to capture certain program metrics like “number of loops” and “number of variables”; Ability to combine similar features to achieve a single task in a better way (Hybrid capability)

**Algorithm for the technique and its application in JPlag:**

For this review, we consider the Greedy-String-Tiling (GST) algorithm. This algorithm was used in the development of JPlag, a Plagiarism Detection System that is discussed below.

The GST algorithm identifies the longest plagiarized sequence of substrings from the text of the source document and returns the sequence as tiles (i.e., the sequence of substrings) from the source document and the suspicious document.

In JPlag, all programs to be compared are parsed (or scanned, depending on the input language) and converted into token strings. These token strings are compared in pairs for determining the similarity of each pair. During each such comparison, JPlag attempts to cover one token stream with substrings (“tiles”) taken from the other as well as possible. The percentage of the token streams that can be covered is the similarity value. The front-end conversion of compared programs to token strings is the only language-dependent process in JPlag. As a rule, tokens should be chosen such that they characterize the essence of a program (which is difficult to change by a plagiarist) rather than surface aspects. For instance, whitespace and comments should never produce a token, since they are the most obvious points of attack. Some sorts of tokens may or may not be considered useful. For instance, the default token set for Java completely ignores expressions except for assignments and method calls.

When comparing two strings *A* and *B* with GST, the aim is always to find a set of substrings that are the same and satisfy the following rules:

* Any token of *A* may only be matched with exactly one token from *B*. This rule implies that it is impossible to completely match parts of the source text that have been duplicated in a plagiarized program.
* Substrings are to be found independent of their position in the string. This rule implies that reordering parts of the source code is no effective attack.
* Long substring matches are preferred over short ones, because they are more reliable. Short matches are more likely to be spurious.

**Technique’s Benefits and similarity detection Quality:**

* It is easy to implement
* Effective at detecting similarity in source codes (though may be time-inefficient if not optimized)

**Review of Latent Semantic Indexing (LSI) for Source code similarity measure**

LSI is a means of getting the contextual meaning of words from documents or passages by using mathematical and statistical methods. It can be used to show various relationships between word to word, word to passage and passage to passage. LSI can be used to simulate human cognitive phenomena such as developmental acquisition of recognition vocabulary to word-categorization, sentence-word, semantic priming and is closely identical to the way a human might categorize various documents.

LSI has been defined as a fully automatic mathematical/statistical technique for extracting and inferring relations of expected contextual usage of words in passages of discourse. It is not a traditional natural language processing or artificial intelligence program; it uses no humanly constructed dictionaries, knowledge bases, semantic networks, grammars, syntactic parsers, or morphologies, or the like, and takes as its input only raw text parsed into words defined as unique character strings and separated into meaningful passages or samples such as sentences or paragraphs.

It involves the use of Singular Value Decomposition. This in mathematics means the factorization of a matrix using 3 matrices. It can be shown that for any *m* X *n* matrix, M, it can be expressed as a product of three vectors

**M = UEVT**

where,

M = the matrix,

 U = an *m* X *m* matrix

 VT = transpose of V of order *n* X *n*

**Implementation of LSI**

A simple implementation can be done as follows:

* First you express the texts as a matrix. The rows will consist of unique keywords while the columns will represent the documents from which they were extracted
* Next, the cells in the matrix will hold the counts of the keywords in their documents (ie columns)
* The counts are usually modified so that so that rare words are weighted more heavily than common words. A popular weighing technique is the TFIDF (ie Term Frequency – Inverse Domain Frequency)
* Next, SVD is applied to the generated matrix using an appropriate dimension

**Benefits of LSI in code similarity measure:**

LSI has the following benefits that make it a technique of choice in measuring source file similarities:

1. LSI overcomes two of the most problematic constraints of Boolean keyword queries: multiple words that have similar meanings (synonymy) and words that have more than one meaning (polysemy).
2. LSI performs automated document categorization.
3. Because of its strict mathematical approach, LSI is inherently independent of language.
4. As opposed to being restricted to only words, LSI can also process arbitrary character strings.

**NICAD: Accurate Detection of Near-Miss Intentional Clones, Using Flexible Pretty-Printing and Code Normalization**

NiCad is a tool that applies a novel approach to clone detection. It uses a source transformation system, TXL, and an efficient text line comparison technique. Although it could be very useful at detecting a lot other types of clones, this review mainly presents the results related to near-miss intentional clones, at function level.

This tool’s approach is a multi-pass one, which is parser-based and language-specific, using simple text line comparison to achieve good time and space complexity. It uses flexible pretty-printing, code normalization and filtering together with comparison algorithm and clustering of potential clones for achieving a good level of clone detection and good memory and time usage.

NICAD has 6 main components: Extraction of Potential Clones, Flexible Pretty-Printing, Flexible Code Normalization, Flexible Code Filtering, Comparing the potential Clones, and, Output Generation.

The Extraction of Potential Clones step has as purpose, extracting possible candidates for clones that respect the minimal length of the clone. This minimal length is important for reducing the amount of further computations, and for presenting a more readable result at the end (ignoring clones that are too small for being useful). The implementation of this step is pretty straightforward, by using TXL extract function [^]. This will return a set of all embedded instances of one grammatical type, into a bigger one (function definitions from the entire source code for example).

Flexible Pretty Printing is a novel approach in clone detection. It uses the TXL’s agile parsing, for breaking different parts of a statement into several lines that will be analysed by text comparison component. This allows different parts of a statement to be compared at different granularities. So, an item to be compared may contain one token or several tokens, according to the used pretty-printing rules. This is an advantage above the other clone detection tools, since parts of a statement can be detected as clones. Also, because of using TXL, this statement breaking is pretty easy to implement, by using just a modified version of the grammar.

Flexible Code Normalization is possible because of the TXL’s replacement rules. This allows normalization of different parts of a grammar element, removing the limitation to global replacement from other tools. By using TXL patterns, NICAD provides flexibility in applying the normalization, by choosing only to normalize within certain type of statement or within a certain level of nesting.

Flexible Code Filtering is done with the help of TXL’s rules, being simple and efficient. All it has to do is to replace the uninteresting statements (defined by using a TXL pattern), with empty ones. The comparison of the potential clones is done with a Longest Common Subsequence algorithm, for comparing the text lines of potential clones. This algorithm will compare the pretty-printed and normalized sequences of text lines from clone, as items. Then it determines the number of unique items in each potential clone. Then it computes the percentage of unique items for each potential clones, as number of unique items / total number of items. If this percentage is below a certain threshold, then the sequences are considered to be clones of each other.

NICAD provides the results in two different representations: the textual report of the clone class information, where each clone class is shown with the corresponding name and line numbers, or the visual representation, which generates an HTML page showing the first code segment as an exemplar for each class clone. The experimental results of this tool so far, show it to be pretty good. The tool was used for finding clones into **Abyss** and **Weltab**.