Clone Wars

VIKTÓRIA FÖRDŐS, ELTE-Soft Nonprofit Ltd.
MELINDA TÓTH and TAMÁS KOZSIK, Eötvös Loránd University

Code clones are unwanted phenomena in legacy code that make software maintenance and development hard. Detecting these clones manually is almost impossible, therefore several code analyser tools have been developed to identify them. Most of these detectors apply a general token or syntax based solution, and do not use domain specific knowledge about the language or the software. Therefore the result of such detectors contains irrelevant clones as well. In this paper we show an algorithm to refine the result of existing clone detectors with user defined domain specific predicates to preserve only useful group of clones and to remove clones that are insignificant from the point of view defined by the user.

Categories and Subject Descriptors: D.2.3 [**Software Engineering**] Coding Tools and Techniques; D.2.7 [**Software Engineering**] Distribution, Maintenance, and Enhancement; F.2.2 [**Analysis of algorithms and problem complexity**] Nonnumerical Algorithms and Problems

General Terms: Languages, Design

Additional Key Words and Phrases: Grouping, Filtering, Clone detection, Erlang, Suffix tree, Static program analysis

1. INTRODUCTION

Code clones, the result of the ”copy&paste” programming technique, have negative impact on software quality and on the efficiency of the software maintenance process. Although copying may be the fastest way of creating a new feature, after a while it is really hard to detect and maintain the multiple instances of the same code snippets.

Based on static source code analysis, clone detectors try to identify code clones automatically. Several clone detectors exist [Roy et al. 2009] applying different techniques to select the clones. These techniques include string, token, syntax and also semantics based approaches.

In the context of the Erlang programming language [Armstrong 2007], there are three clone detectors [Li and Thompson 2009; Fördős and Tóth 2014b; 2013] implementing different techniques to select duplicated code. Although the clones identified by these techniques can be considered duplicates, some of them are *irrelevant* in certain points of view. The filtering system of Clone IdentifiErl allows users to tailor the result in different ways using domain specific knowledge about the language.

This filtering technique can be easily applied on duplicate code detectors that yield clone pairs [Fördős and Tóth 2014b; 2013]: it simply leaves out the pairs which do not fulfil the requirements. When a clone detector groups the identified clones [Baker 1996; Koschke 2012; Fördős and Tóth 2014a], the result is more comprehensible, but makes the filtering less straightforward. Filtering out some part of a group of clones results in smaller groups of clones. Sometimes smaller means that we have less group members, in other cases we have smaller clones – or both.

This work is supported by Ericsson–ELTE-Soft–ELTE Software Technology Lab.
Author's address: V. Fördős, 1/C Pázmány Péter sétány, 1117 Budapest, Hungary; email: f-viktoria@elte.hu; M. Tóth, 1/C Pázmány Péter sétány, 1117 Budapest, Hungary; email: tothmelinda@elte.hu; T. Kozsik, 1/C Pázmány Péter sétány, 1117 Budapest, Hungary; email: kto@elte.hu.

In: Z. Budimac, T. Galinac Grbac (eds.): Proceedings of the 3rd Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications (SQAMIA), Lovran, Croatia, 19.-22.9.2014, published at http://ceur-ws.org.

In this paper we show a general, language independent algorithm to refine the result of existing clone detectors that produce groups of clones. We apply domain specific predicates to the clones to filter out useful groups of clones from different points of view. For example, the clone elimination process can be simplified by removing clone instances that are difficult to be eliminated. The filtering system can be also used to exclude those clones from the result that are duplicates of any given exceptions. For instance, consider that the generated source code should be removed from the results. We also emphasize here that maintenance time can be decreased by focusing on the clones that cause the hardest problems, so the developer can work on the most useful maintenance tasks.

2. RELATED WORK

Clone detection is a wide field of research. Here we focus on filtering and grouping techniques.

Several detectors for duplicates exist, but only a few of them concentrate on functional languages, such as [Brown and Thompson 2010] developed for Haskell, and Wrangler [Li and Thompson 2009] for Erlang. We have proposed Clone IdentifiErl [Fördős and Tóth 2013] for Erlang, which is an AST/metric based approach. We have also published a purely metric driven algorithm [Fördős and Tóth 2014b] that characterises the Erlang language by using software metrics to identify clones in Erlang programs. In Clone IdentifiErl, a new standalone extensible filtering system has been introduced to filter out irrelevant clones, whilst in [Fördős and Tóth 2014b] we have given a filtering system that is capable of removing both irrelevant and false positive clones. The papers [Fördős and Tóth 2014b; 2013] argued for the necessity of this step, and presented a domain-specific implementation for Erlang.

Earlier, Juergens and Göde [2010] have proposed an iterative, configurable clone detector (ConQAT) containing a filtering system. ConQAT can remove repetitive generated code fragments and overlapping clones by iteratively reconfiguring and rerunning its initial clone detector.

Different clone detection techniques have been used by known detectors. Some of them, e.g. the suffix-tree algorithm [Baker 1996], form groups from the resulting clones. On the other hand, there are algorithms that produce clone pairs. In this latter case, it is also possible to group the results, but this has an additional computational overhead. For example, paper [Fördős and Tóth 2014a] shows a general and broadly usable method to group the result of clone detection algorithms.

Although significant research has been carried out in this area, grouping and filtering in one step is a novel technique. The method, presented in this paper, does not conflict with the already existing techniques and tools: it is an additional tool to refine existing results.

3. IMPROVING CLONE DETECTION

There are many algorithms for detecting code duplicates. They differ in accuracy as well as in execution time. Our goal is to improve accuracy without compromising efficiency. In this paper, we propose a standalone, and also a language independent, approach that can be used after any duplicated code detector to tailor its results. Here, we also present how it can be configured to facilitate accurate clone detection in Erlang programs.

Clone detectors result in either clone pairs or clone groups. Regardless of the format of the result, our algorithm can improve it until pairs can be considered as groups of two-elements. Therefore, our algorithm is defined to work with *initial clone groups*. It examines each initial clone group whether all elements of the group are "relevant" enough. The goodness of an element is judged by easily replaceable *filters*, thus the behaviour of the algorithm can be customised to fit various purposes. Our algorithm decomposes a group into sub-groups on which all the filters hold, and removes irrelevant elements from the result.

The algorithm provided here assumes certain properties for initial groups. Namely, each clone in a group must have the same amount of building blocks. In our Erlang implementation, the building block is a *top-level expression*, where a top-level expression refers to the main expression or to a sequence of main expressions making up a function clause. There are many clone detectors [Roy et al. 2009] that produce initial clone groups for which the required property holds. Note that a well-known detection technique, the suffix tree based algorithms [Baker 1996; Koschke 2012], provide initial clone groups with the required property. A clone detector employing suffix tree [Gusfield 1997] has been implemented in Wrangler [Li and Thompson 2009] (and also in RefactorErl [RefactorErl project 2014]), and we will use this kind of initial groups for the practical evaluation of our approach in this paper. Before we detail our algorithm, we briefly review the general clone detection technique that uses suffix tree.

3.1 Suffix tree

Usually, clone detectors that use suffix tree are token-based algorithms. Tokens of the analyzed program are mapped to words of a formal language based on the token kind. For instance, identifiers and literals can be mapped to ‘a‘, the `begin` keyword is mapped to ‘b‘, the `case` keyword is mapped to ‘c‘ and so on. The suffix tree is constructed from using the transformed tokens. The groups of initial clones are gathered as subtrees from the entire suffix tree. This step requires $\mathcal{O}(n * \log n)$ (n denotes the number of tokens) steps in the worst case.

The main advantages of this technique are the low computational cost and the compact result, because it demands no further grouping. Several duplicated code detectors [Baker 1996; Koschke 2012] use this algorithm as their initial clone detectors, because suffix tree based clone detection is a general technique that can be easily applied to detect clones in any programming language.

However, its general applicability implies its weak points. It is a token-based detector with no built-in knowledge of programming languages, thus several clones forming no valid syntactical unit may appear in its result. These clones need to be further cut to meet the syntactical rules of the programming language in which the clones were implemented. Hereupon, it produces several useless clones that consist of a few tokens and have nearly no syntactic characteristics. Last but not least, finding gapped clones becomes impossible, because the algorithm of suffix tree cannot deal with these clones.

3.2 Filtering

Our algorithm examines each initial clone group whether its elements satisfy the required properties described by the user defined filters. Filters are applied to the building blocks of the clone instances that belong to the same group. Filters can be chosen arbitrarily to fit any purpose.

Our algorithm allows developers to concentrate only on important clones by removing irrelevant clones from the result, as Clone IdentifiErl does. Our algorithm can also come to rescue, if the result needs to be cleaned by excluding elements that are required to be duplicated. For instance, consider constraints originating from business logic, or the cases when a function acts as a bridge that connects two applications. Expressions referring to these functions are obviously clones, but they are necessarily present. For another example, consider that the examined source code contains a parser (e.g.: `yecc`) generated source code which can be excluded from the clone groups by using our algorithm. To best of our knowledge, no Erlang specific approach can handle such exceptions.

In this paper, we show how our algorithm can ease the elimination process by using Erlang specific filters. Although a suffix tree based detector is being employed to produce the initial clones, our implementation already ensures that the initial clones of a group are real syntactic clones that differ only in the used identifiers and operators. Thus, we only want to check that the following two predicates hold for a group to ease the elimination process.

—The elements of the group refer to nearly the same set of functions.

```
% First instance
?Query:exec(Nd, ?Query:seq([?Expr:clause(),
                            ?Clause:form(),
                            ?Form:func()]))
```

```
% Second instance
?Query:exec(RecNode, ?Query:seq([?Rec:file(),
                                 ?File:included(),
                                 ?File:module()]))
```

Fig. 1. A clone whose instances refer to different functions

$$\begin{array}{c c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \left(\begin{array}{ccccc} a & x & a & y & a \\ b & c & b & c & b \\ d & e & d & e & z \end{array}\right) \end{array}$$

Fig. 2. Record types referred by expressions in five three-unit long clone instances of a group

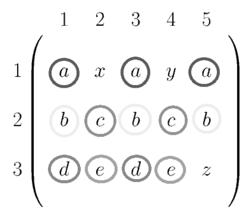

Fig. 3. Maximal sized groups containing one-unit long clone instances

—The elements of the group refer to the same record definitions.

The first property excludes clones whose elements call mostly different functions, because the functionalities implemented in these clones are likely to be independent. Thus, the elimination of these clones is not a high-priority task. For instance, consider Figure 1. The latter property is greatly Erlang specific. Note that these filters should only be replaced when tailoring the algorithm to report clones written in another programming language.

3.3 Grouping & filtering in one step

In this section we briefly explain how to process an initial clone group by filtering and regrouping clones. The algorithm decomposes a group containing m pieces of n-unit long clone instances into subgroups based on filtering results.

The original group can be best understood as an expression matrix of size $n \times m$: a column in this matrix is a clone instance, i.e. a sequence of n (top-level) expressions appearing in the program that was categorized by the initial grouping as a clone of the other columns in the same matrix. A row in the matrix contains m occurrences of a "similar" expression. For efficiency of the initial clone detection phase, this similarity may be too permissive. We can design more specific "filters" to express domain-specific knowledge about "relevant" clones, by considering two expressions similar in a more restrictive manner. For instance, we can introduce a filter which considers two expressions different if they refer to different record types – even if they were declared similar by the initial clone detection.

A subgroup is formed as an intersection of some selected rows and columns from the original expression matrix. Columns of a subgroup are clones that are relevant from the point of view of the applied filtering mechanism. Our algorithm will try to find maximal sized sub-groups, with elements consisting of as many units as possible.

For the sake of the example, assume that we characterise expressions based on the referred record types. Let us denote an expression with 'a' if it refers to a record type 'a'. In Figure 2, a characterisation of an initial clone group can be seen. The group contains five (i.e.: $m = 5$) three-unit long clone instances (i.e.: $n = 3$). The elements of the matrix represent the records referred by the expressions of the clones; the third element of the first row is an expression referring to record 'a' (only). Note that only two of the initial clones in the group are considered relevant by this filtering: column 1 and column 3. Furthermore, our algorithm will identify a shorter clone consisting of two top-level expressions *c;e* in columns 2 and 4 as well.

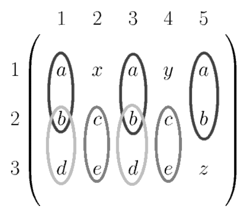

Fig. 4. Joining sub-groups

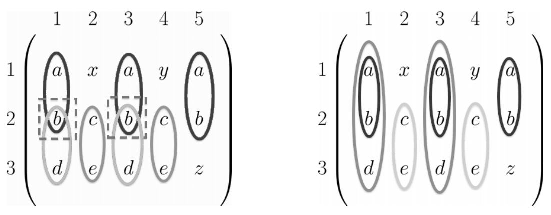

Fig. 5. Finding sub-groups to be glued

We want to maximise both the length (number of columns) and the size (number of rows) of every constructed sub-group. These properties are orthogonal to each other, therefore the maximisations of them impede each other.

We propose an iterative algorithm. We start by identifying sub-groups having one-unit long clone instances. For our example, such sub-groups are shown in Figure 3: in each row, the elements that belong to the same sub-group are painted using the same shade of grey. Obviously, we create maximal sized sub-groups. In the third row, for instance, we could identify two sub-groups (one for d and another one for e), both containing two columns. In the first row, however, we have a sub-group with three columns.

Next, we try to improve the other dimension, i.e. to lengthen the elements of sub-groups. This goal is achieved in two steps. First, we try to join the previously determined sub-groups; here care must be taken not to lose any existing maximal sized sub-groups. We can join two sub-groups if there are no rows between them (clones are continuous blocks of top-level expressions), and they share at least two columns (i.e. at least two clones contain the matching expressions). Furthermore, if there is a clone instance in any of the to-be-joined sub-groups that is not included in the newly created sub-group, then the original sub-group containing this clone instance must be preserved (otherwise it can be thrown away). We will come back to this *covering* problem soon.

Joining is illustrated in Figure 4. Note that matrix elements may belong to multiple sub-groups, as happened with the bs in the second row of the expression matrix. We could join those bs with the as of the first row, as well as with the ds of the third one.

In the second step of sub-group lengthening, we iteratively glue overlapping sub-groups together. Check the left half of Figure 5: two groups (*a;b* in the first two rows and *b;d* in the last two rows) are glued together based on the overlapping in the second row (dashed rectangles). Naturally, it is required that glued sub-groups have at least two common columns: without that they would not be clones. Here the sub-groups share the first and the third columns.

Again, if a clone instance that belongs to any of the input sub-groups is not present in the glued sub-group, then its containing sub-group must be preserved. (We will refer to this phenomena as the new sub-group *not covering* the old one.) This can be observed in the right half of Figure 5. After constructing the new sub-group *a;b;d* in columns 1 and 3, we can drop sub-group *b;d*, but we must preserve the sub-group *a;b* in the first two rows, since it contains the third column, which is not included in the new sub-group, and hence the new sub-group does not cover the original *a;b* sub-group.

The gluing step is repeated until there are no sub-groups that can be glued together. Then the algorithm terminates, and outputs the determined sub-groups as a refined grouping of clones.

4. FORMAL DESCRIPTION

Now we define our filtering&grouping algorithm more precisely. The algorithm operates on a single clone group, represented as an expression matrix of size $n \times m$. Each column represents a sequence of top-level expressions in a function clause (a clone instance), and a row corresponds to similar expressions with respect to an initial clone detection algorithm.

$$G \in \mathcal{E}^{n \times m}$$

For filtering out irrelevant clones, we will use a reflexive and symmetric (but not necessarily transitive) binary relation over expressions.

$$f \subseteq \mathcal{E} \times \mathcal{E}$$

Our algorithm takes a clone group G and a filtering relation f, and produces a set of subgroups, which refines the grouping G.

$$G, f \mapsto \{G_1, G_2, \dots G_r\}$$

Each subgroup G_i represents a clone group made up of m_i clone instances, each instance having length n_i, where $1 \leq n_i \leq n$ and $2 \leq m_i \leq m$.

$$G_i \in \mathcal{E}^{n_i \times m_i}$$

A subgroup selects a block of rows and some of the columns of the original expression matrix. The initial clones represented by the selected columns in the subgroup contain an expression sequence (the selected rows) which is accepted as a “relevant” clone.

We can represent a subgroup with a “selection” s, relative to an initial group G. The block of rows selected by s is denoted by $s.r$, where $s.r.\ell$ is the lower, and $s.r.u$ is the upper bound of the selection. The set of columns selected by s is denoted by $s.c$. (To improve the efficiency of the algorithm, $s.c$ can be represented as an ordered list of numbers.)

$$s = (r, c) \quad \text{where } r = (\ell, u), \quad \ell \in [1..n], \quad u \in [\ell..n], \quad c \subseteq \{1, \dots, m\}$$

The algorithm is defined as two steps (Sections 4.1 and 4.2, respectively) followed by an iteration of a third step (Section 4.3). No more than $n - 2$ iterations of the third step are needed; the algorithm can terminate earlier if fixed point is reached.

4.1 One unit long clone instances

The first step of the algorithm produces S_1, a set of selections of the initial group.

$$G, f \mapsto S_1$$

Each clone instance of each selection in S_1 has length 1 (these clone instances are formed from only one expression). This is the only step of the algorithm where filtering takes place, and predicate f is used. All pairs formed from the elements of each selection in S_1 satisfy predicate f. In the subsequent steps we shall maximize both the length of reported clone instances, and the size of the subgroups.

$$S_1 = \bigcup_{i=1}^{n} \Big\{ \ ((i,i), c) \ \Big| \ c \in \mathrm{MaxProperCliques}\big(\mathrm{graph}(f, G, i)\big) \Big\}$$

where $\mathrm{graph}(f, G, i)$ is the graph of f regarding to the expressions of the i^{th} row in G with vertices $\{1, \dots, m\}$ and edges

$$\Big\{ (p, q) \ \Big| \ \big(G[i,p], G[i,q]\big) \in f \Big\}$$

and $V \in \mathrm{MaxProperCliques(g)}$ means that V is a clique (the vertices of a complete subgraph) of g which contains at least 2 vertices, and V is not included in a larger clique (i.e. V is inclusion-maximal [Bomze et al. 1999]).

4.2 Joining clone instances

The second step of the algorithm takes clone subgroups containing one unit long clones, and try to join subgroups.

$$S_1 \mapsto S_2$$

Joining can be defined in two steps. First, we introduce S'_1 as follows.

$$S'_1 = \Big\{ \big((\ell_1, u_2), c_1 \cap c_2\big) \;\Big|\; \big((\ell_1, u_1), c_1\big) \in S_1, \big((u_1 + 1, u_2), c_2\big) \in S_1, |c_1 \cap c_2| > 1 \Big\}$$

Let the binary relation $\underline{\text{covers}}$ over selections be defined as a partial order in the following way: s_1 $\underline{\text{covers}}$ s_2 if and only if

$$s_2.c \subseteq s_1.c \;\wedge\; s_1.r.\ell \leq s_2.r.\ell \;\wedge\; s_2.r.u \leq s_1.r.u$$

Finally, we can provide S_2 by combining S_1 and S_2 and eliminating selections that are already covered by other, larger selections.

$$S_2 = S'_1 \cup S_1 \setminus \big\{ s \;\big|\; \exists s' \in S'_1 : s' \;\underline{\text{covers}}\; s \big\}$$

4.3 Glueing clone instances

The third step, which must be repeated until fixed point is reached (which will happen after no more than $n - 2$ iterations) is also described in two steps.

$$S'_i = \Big\{ s \;\Big|\; s_1, s_2 \in S_i,\; s_1.r \;\underline{\text{overlaps with}}\; s_2.r, s.r.\ell = \min(s_1.r.\ell, s_2.r.\ell), s.r.u = \max(s_1.r.u, s_2.r.u), s.c = s_1.c \cap s_2.c, |s.c| > 1 \Big\}$$

where two blocks of rows are overlapping, i.e.

$$(\ell_1, u_1) \;\underline{\text{overlaps with}}\; (\ell_2, u_2) \text{ if and only if } (\ell_1 \leq \ell_2 \leq u_1) \vee (\ell_2 \leq \ell_1 \leq u_2).$$

Now we can define S_{i+1} by removing all the selections from S'_i that are covered by other, larger selections.

$$S_{i+1} = S'_i \setminus \big\{ s \;\big|\; \exists s' \in S'_i : s' \;\underline{\text{covers}}\; s \big\}$$

When the iteration of this third step reaches fixed point, the last set of selections, S_t can be used to determine the set of subgroups returned by our algorithm. For each selection $((\ell, u), c) \in S_t$, we yield a subgroup of size $(u - \ell + 1) \times |c|$, containing the intersection of the selected rows and columns of the initial expression matrix.

5. CONCLUSIONS

In this paper, we proposed a broadly usable filtering algorithm that quickly removes those clones from the results that are insignificant from the point of view defined by the user. The proposed algorithm is language independent, thus the results of many duplicated code detectors can efficiently be improved. By removing irrelevant clones, the maintenance costs can be decreased, because the programmers need to only deal with important issues.

In this paper, we defined rules that are specialised for easing the clone elimination process in Erlang programs. We discussed the underlying ideas, and we also gave a formal description of our algorithm.

We note that we successfully evaluated the realisation[1] of the algorithm and assessed the results. All of our goals were reached; clones that are hard to eliminate are not present in the results. The filtering

[1]The authors would like to thank to Bence Szabó for the implementation.

phase requires only a small extra computational cost that is infinitesimal. Moreover, it removes clones that are insignificant from the point of view defined by the user. Thus, the algorithm quickly cleans the result and helps programmers focus on only important cases.

Future work will consist of evaluating the proposed algorithm by using initial clones reported by different clone detectors and studying and comparing the results of these test runs. Differences in the results will indicate areas for future study.

REFERENCES

Joe Armstrong. 2007. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf.

Brenda S. Baker. 1996. Parameterized Pattern Matching: Algorithms and Applications. *J. Comput. System Sci.* 52, 1 (1996), 28 – 42. http://www.sciencedirect.com/science/article/pii/S0022000096900033

Immanuel M. Bomze, Marco Budinich, Panos M. Pardalos, and Marcello Pelillo. 1999. The maximum clique problem. *Handbook of Combinatorial Optimization (Supplement Volume A)* 4 (1999), 1–74. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.6221

Christopher Brown and Simon Thompson. 2010. Clone Detection and Elimination for Haskell. In *PEPM'10: Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, John Gallagher and Janis Voigtlander (Eds.). ACM Press, 111–120. http://www.cs.kent.ac.uk/pubs/2010/2976

Viktória Fördős and Melinda Tóth. 2013. Identifying Code Clones with RefactorErl. In *Proceedings of the 13th Symposium on Programming Languages and Software Tools, ISBN 978-963-306-228-9*. Szeged, Hungary, 31–45.

Viktória Fördős and Melinda Tóth. 2014a. Comprehensible presentation of clone detection results. In *Proceedings of the 4th Symposium on Computer Languages, Implementations and Tools (accepted)*.

Viktória Fördős and Melinda Tóth. 2014b. Utilising the software metrics of RefactorErl to identify code clones in Erlang. In *Proceedings of 10th Joint Conference on Mathematics and Computer Science (Informatica)*, Vol. LIX. Studia Universitatis Babeş-Bolyai, Cluj-Napoca, Romania, 103–118. Issue Special Issue 1.

Dan Gusfield. 1997. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, New York, NY, USA.

Elmar Juergens and Nils Göde. 2010. Achieving Accurate Clone Detection Results. In *Proceedings of the 4th International Workshop on Software Clones (IWSC '10)*. ACM, New York, NY, USA, 1–8. http://doi.acm.org/10.1145/1808901.1808902

R. Koschke. 2012. Large-Scale Inter-System Clone Detection Using Suffix Trees. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference*. 309–318.

Huiqing Li and Simon Thompson. 2009. Clone detection and removal for Erlang/OTP within a refactoring environment. In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation (PEPM '09)*. ACM, New York, NY, USA, 169–178. http://doi.acm.org/10.1145/1480945.1480971

RefactorErl project. 2014. Suffix tree based duplicate code analysis. (July 2014). http://pnyf.inf.elte.hu/trac/refactorerl/wiki/SuffixTreeBasedDuplicateCodeAnalysis

Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Sci. Comput. Program.* 74, 7 (May 2009), 470–495. http://dx.doi.org/10.1016/j.scico.2009.02.007