Incremental Clone Detection

Diploma Thesis

Submitted by

Nils Göde

on

1st September 2008

University of Bremen
Faculty of Mathematics and Computer Science
Software Engineering Group
Prof. Dr. Rainer Koschke

Acknowledgements

I would like to thank the members of the Software Engineering Group for discussing my ideas and giving me feedback. Special thanks go to Rainer Koschke and Renate Klempien-Hinrichs for supervising this thesis. Furthermore, I would like to apologize to my family and friends, who did not get the share of my time that they deserved.

Declaration of Authorship

I declare that I wrote this thesis without external help. I did not use any sources except those explicitly stated or referenced in the bibliography. All parts which have been literally or according to their meaning taken from publications are indicated as such.

Bremen, 1st September 2008

. . . .
(Nils Göde)

Abstract

Finding, understanding and managing software clones — passages of duplicated source code — is of large interest, as research shows. However, most methods for detecting clones are limited to a single revision of a program. To gain more insight into the impact of clones, their evolution has to be understood. Current investigations on the evolution of clones detect clones for different revisions separately and correlate them afterwards.

This thesis presents an incremental clone detection algorithm, which detects clones in multiple revisions of a program. It creates a mapping between clones of one revision to the next, supplying information about the addition, deletion or modification of clones. The algorithm uses token-based clone detection and requires less time than using an existing approach to detect clones in each revision separately. The implementation of the algorithm has been tested with large scale software systems.

Contents

1 Introduction

To be faster or not to be faster:
That is the question!

1.1 Software Clones

Duplication of source code is a major problem in software development for different reasons. The source code becomes larger [MLM96] and more difficult to understand, as copied passages have to be read and understood more than once [BYM$^+$98]. A very serious issue arises from errors, *bugs*, found in any of the copies. In a lot of cases, copies are not created to be identical parts of code, but serve as basic structure for the new code to be written. This means that slight changes are made to copies like renaming identifiers or changing constants [MLM96]. The more changes are made to the copies, the harder they become to trace. If bugs are found in the unchanged part, there are no sophisticated means of retrieving all other copies which need to be changed as well [Bak95, DRD99, Joh94, KKI02, Kon97]. This might lead to inconsistent changes of parts which are actually meant to be equal [KN05, Kri07].

Apart from the negative side, there is quite a diverse range of more or less comprehensible reasons for copying source code. The first striking reason is simplicity. It is often easier to copy and maybe slightly modify an existing portion of code than rethinking and writing things from scratch. This reduces the probability of introducing new bugs, assuming the original code is known to work reliably [Bak95, BYM$^+$98, DRD99, Joh94]. In highly optimized systems, the overhead of additional procedure calls resulting from an *extract method* refactoring [Fow99] might not be acceptable. Parts of the code which are frequently executed are repeated instead of being abstracted into a new function [Bak95, BYM$^+$98, DRD99, KKI02]. Architectural reasons, which include maintenance issues and coding style, might also require the repetition of code [Bak95, MLM96, BYM$^+$98]. In addition to intentionally duplicated code, passages might be accidentally repeated. Unintentional repetition might emerge from frequently used patterns of the programming language or certain protocols for using libraries and data structures [BYM$^+$98, KKI02]. Finally, non-technical issues can lead to code duplication. If a programmer is assessed by the amount of code he or she writes, it is quite tempting to copy portions of code [Bak95, DRD99].

Although the negative impact of copied source code passages, *clones*, is not yet proven and several works discuss this topic contrarily [KG06, KSNM05, Kri07, LWN07], information about the presence and evolution of clones is of great interest to study their influence on a system. Several approaches have been presented for finding and analyzing clones. The following section gives an overview over different methods used in software clone detection.

1.2 Detecting Clones

Preventing duplication of code right from the start is a rather illusive objective as there are diverse reasons for copying passages of source code as mentioned above. Instead, a lot of effort is spent on finding clones in existing source code using a clone detection algorithm. The term *software clone detection* summarizes all methods that focus on finding similar passages of source code in a software system.

Different algorithms operate on different abstractions of a program to find clones. The most basic abstraction is the source code itself. Certain approaches use the source text, with or without normalizing it, and find clones by applying textual comparison techniques [DRD99, Joh94, WM05]. Without normalizing the source code, any yet so small difference like whitespace or comments in source passages can prevent tools from reporting clones. Though textual comparison is relatively fast and easy to apply, the quality of the results might lack from disregarding any syntactic or semantic information of the source code.

Other methods operate on the token string (see Section 2.1) that is produced from the program's source code by running a lexer on it. By knowing something about the syntax of the program, these methods are able to abstract from certain aspects of the source text like whitespace and text formatting in general. Within the string of tokens, similar substrings are searched and reported as clones [Bak95, CDS04, KKI02, LLM06]. The advantage of token-based clone detection is that it performs very well, as the source code needs only to be converted into a string of tokens. The detection is also language independent as long as there exists a lexer for the language the source code is written in. Another benefit is that the source code does not need to be compilable and the detection can be run in any stage during the development of a program.

Yet other approaches search for clones based on the *AST (Abstract Syntax Tree)* of the program. Within the AST, subtrees are compared against each other and sufficiently similar subtrees are reported as clones. As the number of comparisons can rapidly grow fast due to an AST with n subtrees requiring n^2 comparisons, different criteria are used to select trees which need to be compared against each other [BYM$^+$98, EFM07, JMSG07, Yan91]. Other methods use the AST just as an intermediate result and do further processing in order to find clones [KFF06, WSvGF04]. As the syntactic structure of a program is represented inside the AST, tree-based approaches are able to report clones which are usually not detected by the previous methods. These include for example commutative operations where the order of the two operands has been inverted. Although the usefulness of reported clones is increased, performance is a critical issue for tree-based detection.

Apart from the AST, the *PDG (Program Dependency Graph)* might be considered for drawing conclusions about clones [KH01, Kri01]. A PDG represents the data flow and control flow inside a program. Inside the PDG, similar subgraph structures are used to identify clones. The additional information that is obtained from the PDG allows to improve the quality of reported clones. On the other hand, creating a PDG and searching clones within it can be very costly. Like tree-based detection, graph-based approaches depend on the programming language of the program analyzed.

A rather different approach to finding clones is based on metrics retrieved for syntactic units of the program [DBF$^+$95, MLM96, PMDL99]. If certain units are equal or similar in their metric values, they are supposed to be clones. This is based on the assumption, that if two units are equal in their metric values, they are equal — or at least sufficiently similar — to be reported as clones. In any case, metrics have to be chosen carefully in order to retrieve useful results.

All these approaches have advantages and drawbacks. Which method yields the best results for a given scenario depends to a large extend on the specific application. A comparison of selected clone detection methods which operate on different abstractions of a program can be found in [Bel07].

Though being quite diverse, all these approaches have in common that they are all targeted at analyzing a single *revision*[1] of a program. For many applications this is the desired behavior, but still there are scenarios which require more than the analysis of a single revision. Considering all questions aimed at the evolution of software clones requires analyzing more than one revision of a program. Current approaches trying to answer evolutionary questions about software clones usually start by analyzing each revision of the program separately, utilizing one of the existing detection techniques. After the clones have been identified for each revision, clones of different revisions are matched against each other based on some definition of similarity [ACPM01, KSNM05, Kri07]. If a clone in revision i is adequately similar to a clone in revision $i-1$, it is assumed that the clone is the same. Some of

[1]Within the context of this thesis, a *revision* is seen as a state of program's source code at a specific point of time

these methods are summarized in Section 4. There are however two major issues concerning these approaches.

- Although running adequately fast, all clone detection algorithms still need a noticeable amount of time to produce their results. If a huge amount of revisions of a program is to be analyzed, the time t_{all} required to get results, is a multiple of the time needed to process a single revision t_{single}. Assuming that the time t_{single} is approximately the same for every revision, the overall time to analyze n revisions is

  $$t_{all} \approx n \cdot t_{single}$$

  The assumption is, that a lot of processing steps are repeated redundantly. Intermediate results which might be reused in the analysis of the next revision are discarded and need to be recomputed for every revision, causing an unnecessary overhead. This results in large parts of the program to be analyzed over and over again for each revision, although most parts of the source code did not change at all.

- If every revision of the software is analyzed on its own, the results are independent sets of clones. An important thing missing is the mapping from the clones of one revision to the clones of the next revision. The information about the changes that happened to each single clone is not provided and has to be calculated later.

Figure 1 shows conventional clone detection applied to multiple revisions of a program. Assuming that changes between revisions i and $i + 1$ stay in a limited range, many calculations are done redundantly. Furthermore, the mapping between clones of revision i and $i + 1$ is not created by the algorithm as every revision is analyzed on its own.

The issues mentioned above and the fact that more and more questions are directed towards the evolution of clones [HK08] suggest a detection algorithm that is designed to analyze more than one revision of a program and tries to exploit this attitude as much as possible. Having such an algorithm is desirable as it would most likely save huge amounts of time when analyzing many revisions of a program. Furthermore, it would make clones traceable across revisions and allow analyses of the evolution of clones. This in turn would help answering the question of the harmfulness of clones. The answer is important, because if clones are not to be considered harmful, then the effort spent on finding and removing them is not legitimate.

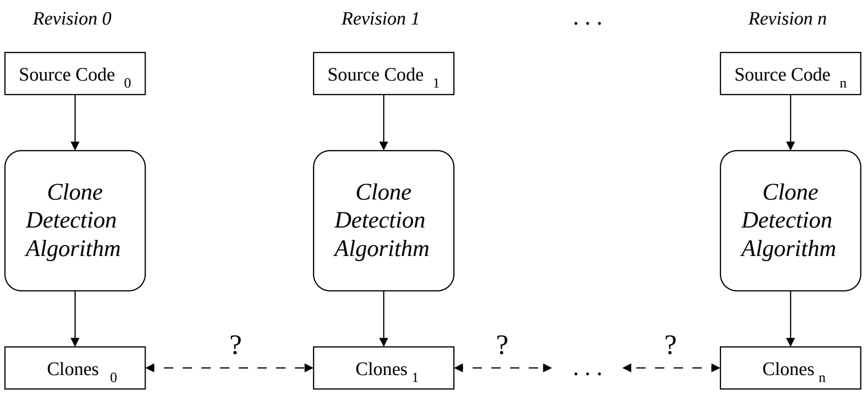


Figure 1 – Conventional clone detection applied to multiple revisions of a program. Calculations are unnecessarily repeated and the mapping between clones is unsure. Solid arrows indicate input and output of the algorithm. Dashed arrows represent the mapping between two clone sets.

Another scenario that might be thought of, is the integration of "on-line" clone detection into an *IDE (Integrated Development Environment)*. Online clone detection reports any changes of clone pairs to the user while he or she is editing the source files of a program. This prevents the user from accidentally introducing new clones or doing inconsistent changes to existing clones. The detection algorithm runs in the background and when the user changes a file, the set of clones is updated accordingly and changes to clones are reported to the user.

1.3 Research Questions

The overall task of this thesis is to develop and implement an incremental clone detection algorithm that requires less time for clone detection in multiple revisions of a program than the separate application of an existing approach. In addition, a mapping between the clones of every two consecutive revisions must be generated. To guide the development and assist the achievement of the task, a number of research questions are given which are to be answered within this thesis.

1.3.1 Multi-Revision Detection

The first part of the task addresses the time t_{all} which is needed to analyze n revisions of a program's source code. The assumption is, that time can be saved by eliminating unnecessary calculations resulting from discarding intermediate results. It is desirable to make $t_{all} < n \cdot t_{single}$ true. Instead of starting from the very beginning, the analysis of a revision should reuse and modify results of the previous revision. This requires an overview over all results which are produced during the clone detection process and assessment of whether they might serve for being reused.

Question 1 – *Which intermediate results can potentially be reused to accelerate multi-revision clone detection?*

It is very unlikely, that reuse can happen straight away, because the intermediate results are not designed for being reused. It is very probable that certain problems arise which must be solved to make the results reusable.

Question 2 – *Which problems arise from reusing intermediate results and how can they be solved?*

After solutions have been given to the problem of reusing intermediate results, a concrete algorithm needs to be presented that puts multi-revision clone detection into practice. The algorithm has to implement the conclusions drawn from answering the previous two questions.

Question 3 – *How does an incremental clone detection algorithm look like?*

1.3.2 Tracing

Apart from improving the performance, clones of one revision are to be mapped to the clones of the previous revision. In the simplest case, clones remain untouched and no change happens to any clone. On the other hand, clones can be introduced, modified, or vanish due to the modification of the files they are contained in. The different changes that can happen to a clone must be summarized.

Question 4 – *Which changes can happen to a clone between two revisions?*

Knowing about the nature of changes, it might be possible to draw conclusions about changed clones from the modification of intermediate results. It is assumed that changes can be derived and do not need to be recreated in a separate phase.

Question 5 – *How can changes to clones be derived while reusing intermediate results?*

If not all changes can be concluded from the modification of results, the remaining changes have to be obtained in a post-processing step. This requires a method that is able to locate a clone from revision i in revision $i+1$.

Question 6 – *How can a clone in revision i be found in revision* $i+1$*?*

1.3.3 Implementation and Evaluation

It is not only required to present the conceptual ideas for incremental clone detection, but also to implement them. The underlying use case for the implementation of an incremental clone detection algorithm is, that given multiple revisions of a program, the clones in each revision are to be detected. It is required, that the incremental algorithm produces its results faster than separate applications of an existing approach. As implementing clone detection from scratch goes far beyond the scope of this thesis, an existing implementation is to be modified. This limits the diversity of theoretical and practical options that can be explored.

The existing tool which serves as a starting point for the implementation is the tool *clones* from the project *Bauhaus*[2]. The performance of the new algorithm's implementation is tested against the performance of *clones* to keep results comparable.

Question 7 – *How does the new implementation perform in comparison to clones, regarding time and memory consumption?*

It is assumed, that no general statement can be made about how much time can be saved by using incremental clone detection. The time most likely depends on different factors which influence the detection process.

Question 8 – *Which factors influence the time required by the new implementation?*

The answers to these questions are gained by developing and implementing the incremental clone detection algorithm. They are given at the respective locations within this thesis.

[2]http://www.bauhaus-stuttgart.de/

1.4 Thesis Structure

This thesis is organized as follows. Section 2 introduces the necessary concepts related to software clone detection. It explains the clone detection process as implemented in the tool *clones*. Section 3 answers the questions related to multi-revision analyses and presents an incremental clone detection algorithm. Section 4 explains changes that can happen to clones and describes how clones can be traced across revisions. Section 5 summarizes the tests that have been run to answer the questions directed at the performance of the implementation. Conclusions and ideas for future development are given in Section 6.

2 Background

This section introduces the necessary concepts for understanding the problem of incremental clone detection and the solution presented in this thesis. Section 2.1 explains tokens and token tables. Section 2.2 explains terms related to software clone detection which are unfortunately used with different meanings by different publications. Therefore it is necessary to have a common understanding of these terms in order to avoid any confusion. Section 2.3 introduces the concept of suffix trees. The relation between clone pairs and suffix trees is outlined in Section 2.4. Finally, Section 2.5 introduces token-based clone detection as implemented in the tool *clones* by describing the major phases.

2.1 Tokens and Token Tables

Throughout this thesis, the word *token* will be used frequently as clone detection using the tool *clones* is based on sequences of tokens forming clones. *Aho et al.* define a token as follows:

> *"[. . .] tokens, that are sequences of characters having a collective meaning."* [ASU86]

A token is an atomic syntactic element of a programming language (i.e. a keyword, an operator, an identifier,. . .). In addition, *clones* also recognizes preprocessor directives as tokens. A sequence of successive tokens is referred to as a *token string*. Each token has certain properties which are accessed in different stages of the clone detection procedure. Among the important properties of a token are the following:

- **Index**: The index of the token inside the token table in which it is contained.
- **Type**: Every token has a type describing the nature of the token. Example types are `+`, `=`, `while` or `<identifier>`.
- **Value**: Tokens which are identifiers or literals have a value in addition to their type. Identifiers have their actual name being the token's value. Literals have the value represented by their string, for instance, literal "`1`" has an integer value 1.
- **File**: The source file which contains the token.

- **Line**: The line in the source file in which contains the token.
- **Column**: The column in the source file in which the token starts. Lines and columns are counted starting from 1.

Tokens are stored inside a *token table* which holds a number of tokens along with their properties. By using a token's index, the token table allows fast access to the token's properties. A token table is created using a *scanner* or *lexer* which translates a source code file into a sequence of tokens. A simple token table is shown in Table 1.

Index	0	1	2	3	4
Type	`<identifier>`	`=`	`<identifier>`	`+`	`<number>`
Value	`a`		`b`		`3`
File	`sample.c`	`sample.c`	`sample.c`	`sample.c`	`sample.c`
Line	`1`	`1`	`1`	`1`	`1`
Column	`1`	`3`	`5`	`7`	`9`

Table 1 – A sample token table for the input `a = b + 3` contained in a file called `sample.c`.

2.2 Fragments and Clone Pairs

When talking about software clones, it is very helpful to have a shared understanding of terms and concepts used to describe clones and relations among them. Far too often, terms like *clone* and *clone pair* are used inconsistently, leading to confusion or requiring additional explanation whenever used.

Although the word *clone* is used in almost every publication related to software clone detection, there exists no definition and no sufficiently common understanding of what is to be treated as a clone and what is not [KAG+07, Wal07]. To avoid any misunderstanding, this thesis will not give any concrete meaning to the word *clone* on its own. Different words are used to describe related concepts. The following sections explain the terms *fragment* and *clone pair* and state how they are used within this document.

2.2.1 Fragment

Throughout this thesis, the word *fragment* refers to a passage in the source code. A fragment consists of a sequence of consecutive tokens. Having a well-defined location, a fragment can be described as a triple

$$fragment = (file, start, end)$$

with *file* being the file in which the fragment appears, *start* the first token[3] in the given file which is part of the fragment, and *end* representing the last token belonging to the fragment. All tokens between *start* and *end* are part of the fragment.

A fragment on its own is not very helpful without knowing whether and which parts of the source code it actually equals. The relation between fragments is described in form of a *clone pair*.

2.2.2 Clone Pair

A *clone pair* is the relation between exactly two fragments that are similar to a certain degree or even identical. A clone pair can be described as

$$clone\_pair = (fragment_A, fragment_B, type)$$

where $fragment_A$ and $fragment_B$ are fragments as described above. In order to express different levels of similarity between two fragments, the clone pair has a *type* in addition to the two source code fragments. Among various descriptions of the similarity between code fragments is the one classifying clone pairs according to four different types. Type 1 to type 3 describe a textual similarity between fragments whereas type 4 refers to fragments which are similar in their semantic.

Type 1: Within a clone pair of type 1, both fragments are exactly equal disregarding comments and whitespace. All tokens at corresponding positions in the two fragments are identical in their type and their value. An example of a type-1 clone pair can be found in Figure 2.

```
int a = 0;
b = a * a;
string = "Peter";
```

(a) $fragment_A$

```
int a = 0;
b = a * a; // Comment
string = "Peter";
```

(b) $fragment_B$

Figure 2 – A clone pair of type 1.

[3]For further processing of fragments — especially by humans — it might be more helpful to give the start and end of a fragment in line numbers. Therefore the token information is converted to line information just before outputting the result.

Type 2: A clone pair of type 2 consists of two fragments whose tokens are identical in their type. In contrast to a type-1 clone pair, the values of identifiers or literals do not need to be identical. Type-2 clone pairs are those where source code has been copied and the names of identifiers have been changed afterwards. Depending on the application, it is sometimes required, that identifiers have been consistently changed, meaning there needs to exist a one-to-one mapping between the identifiers of the first fragment to the ones of the second fragment. *Baker* presented a method for finding type-2 clone pairs with consistent changes, also referred to as *parameterized duplication* [Bak97]. An example clone pair of type 2 is shown in Figure 3.

```
int a = 1;
b = a * a;
string = "Pan";
```

(a) $fragment_A$

```
int d = 1;
d = d * c; // Comment
string = "Pan";
```

(b) $fragment_B$

Figure 3 – A clone pair of type 2 with inconsistent renaming.

In addition to type-1 and type-2 clone pairs, there are two more types which describe more inconspicuous relations between clones. These are given for completeness, but are not considered in this thesis due to their complexity and lack of a concise definition.

Type 3: Type-3 clone pairs combine two or more adjacent clone pairs of the preceding types. These clone pairs may be separated by small code fragments which are not identical. The motivation for type-3 clone pairs is to find pairs which have been modified by inserting or removing tokens from one of the fragments. Such a situation may indicate that a mistake has been found and corrected in one of the fragments, but the other fragment stayed unchanged.

Type 4: Between the fragments of a type-4 clone pair exists an even more vague relation concerning the behavior of the fragments. Code fragments belonging to a clone pair of type 4 carry out similar tasks and are similar in their semantic.

An important property of clone pairs is, that the relation between the two fragments is symmetric, meaning that the existence of clone pair $cp_1 = (fragment_A, fragment_B, type)$ requires the existence of clone pair $cp_2 = (fragment_B, fragment_A, type)$. Though being formally correct, the information contained in both clone pairs is the same and therefore this document abstracts from the order in which the two fragments appear, making the following always true.

$$(fragment_A, fragment_B, type) = cp = (fragment_B, fragment_A, type)$$

To ease the processing of clone pairs, each pair is *normalized* upon creation. In a normalized clone pair, it is fixed which fragment is $fragment_A$ and which is $fragment_B$.

For clone pairs with $type \leq 2$, the relation between the two fragments is not only symmetric, but also transitive. From the existence of pairs $cp_1 = (fragment_A, fragment_B, type)$ and $cp_2 = (fragment_B, fragment_C, type)$ follows, that the pair $cp_3 = (fragment_A, fragment_C, type)$ must exist.

The algorithm described in this thesis relates similar code fragments in terms of clone pairs analogously to the tool *clones*. Still, other relations between code fragments exist. Among them is the relation which groups two or more similar fragments. This relation is usually called *clone class* or *clone community* [MLM96].

2.3 Suffix Trees

Incremental clone detection, as described in this thesis, is based on generalized suffix trees. Therefore it is essential to have an understanding of what suffix trees are and how they represent suffixes of strings of tokens. Many different notions for strings, substrings and suffix trees have been given [Bak93, FGM97, McC76]. This thesis follows the notion presented in [FGM97] in most parts. Although the concepts of strings and suffix trees are described by using strings of characters, the same applies to strings of tokens.

2.3.1 Notion of Strings

A string X which is of length m is represented as $X[0, m-1]$. The character at position i in X is denoted as $X[i]$. Any substring of X, containing the characters at positions $i, i+1, \ldots, j$ is written as $X[i, j]$ with $0 \leq i \leq j < m$. It follows that every substring $X[j, m-1]$ is a suffix and every substring $X[0, i]$ is a prefix of X. The string X contains m prefixes and m suffixes. To ensure, that no suffix is a prefix of any other suffix, the last character at position $m-1$ of the string is globally unique. It does not match any other character from string X and no character from any other string. The unique character is represented as \$, respectively $\$_1$, $\$_2$, …, $\$_n$ if more than one string is used.

2.3.2 Nodes and Edges

A suffix tree is a tree-like data structure which represents all suffixes of a given string X. Disregarding suffix links, which are introduced in Section 2.3.3, the suffix tree is a tree. Every leaf of the suffix tree represents one suffix of X. This makes the suffix tree very useful for solving many problems that deal with strings. Among the many applications, *McCreight* was one of the first to create a suffix tree based upon which equal substrings within a string can be searched [McC76].

Every edge of the suffix tree is labeled with a substring of X. To significantly reduce the space needed by the tree, it is essential to specify the substring by the indices of its first and last character i and j instead of specifying the complete substring. This way, every edge label is a tuple (i, j) referring to the substring $X[i, j]$.

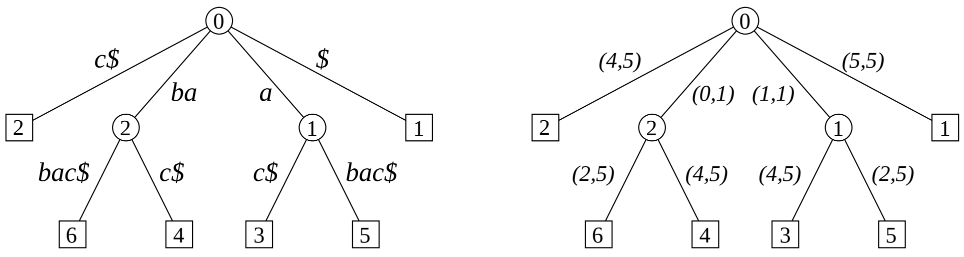
Every edge is connected to two nodes, the *start node* being the node directed towards the root and the other one being its *end node*. An edge is called *internal* if its end node is the start node of at least two other edges. If the end node is not the start node of any other edge, then the edge is called *external*. Note, that it is not possible for any node to have just one edge of which it is the start node. The exception to this rule is the root node of the tree in case of X being the empty string $.

Two edges are called *siblings* if they have the same start node. An important property of suffix trees is, that no two siblings labels start with the same character.

Every node except the root has a *parent edge*, being the edge of which it is the end node. In addition, every node except the root has a *parent node* being the start node of its parent edge. For the root of the tree, parent edge and parent node are undefined. The *path* of a node is the string obtained by concatenating all substrings, referred to by the edge labels from the root to that node. Each node has a *path length*, which is length of its path. The path length for the root is 0.

Like edges, a node is called *internal* if it is the start node of at least two edges, otherwise it is called *external* or *leaf*. Note that every external edge has an external end node and every external end node has an external parent edge. Likewise, every internal edge has an internal end node and every internal node has an internal parent edge. Every external node represents a suffix of the string X which is equal to the path of that node. Therefore, a suffix tree for a string of length m must have m external nodes and m external edges.

A suffix tree for the string $X = babac\$$ is shown in Figure 4. Throughout this document, certain things are to be considered for the visualization of suffix trees. Squares represent external nodes, circles internal nodes. Solid lines are edges connecting the nodes. Though being expressed by start and end index, labels are usually shown as readable substrings to make the understanding of figures easier. Any exceptions to these conventions are mentioned in the description of the respective figure.



(a) Suffix tree with text labels. **(b)** Suffix tree with indices for labels.

Figure 4 – Suffix tree for the string $X = babac\$$. The path length of every node is noted inside the node.

2.3.3 Suffix Links

In order to allow faster construction, suffix trees are augmented with so-called *suffix links* between nodes. If the path from the root to a node represents the substring $X[i, j]$, then the suffix link of that node points to the node whose path represents the substring $X[i + 1, j]$. The suffix link of the root is undefined.

Suffix links do not only help in construction, but also allow for fast navigation inside a suffix tree. Following suffix links helps iterating over all suffixes of a given string from longest to shortest, because every suffix link points to the node whose path represents the next smaller suffix. As suffix links decrease the readability of suffix trees, they will usually not be drawn in figures within this thesis. The suffix tree from Figure 4 augmented with suffix links is shown in Figure 5.

2.3.4 Construction

For long strings and real applications, a fast construction algorithm for suffix trees is needed. Several construction algorithms that require time linear to

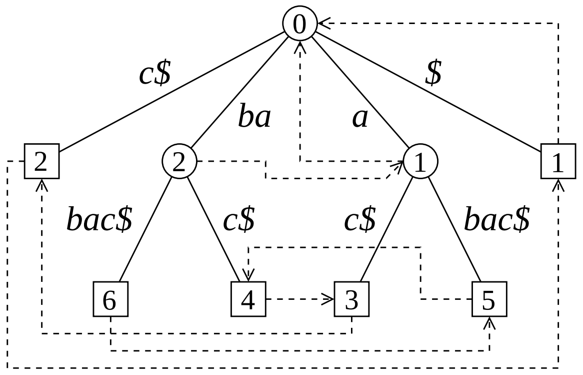


Figure 5 – Suffix tree for the string $X = babac\$$ augmented with suffix links (dashed arrows).

the length of the input string have been presented [Bak97, McC76, Ukk95]. The two algorithms which are implemented and used by the tool *clones* are the following:

Ukkonen: *Ukkonen* presented an *on-line* algorithm for constructing suffix trees [Ukk95]. On-line refers to the algorithms property of processing characters or tokens of the input string from the beginning to the end. At any point of the construction, the suffix tree for the part of the string that has already been processed is available. There is no need to know the complete string upon starting the construction.

Baker (Parameterized Strings): *Baker* introduced an algorithm to find parameterized duplication in strings [Bak97]. In addition, she presents a modified version of *McCreight's* algorithm, that allows construction of parameterized suffix trees for parameterized strings. A parameterized string abstracts from the actual names of identifiers, but still preserves the ordering among them. This constructor is used by *clones* if type-2 clone pairs are required to have a consistent renaming of identifiers.

2.3.5 Applications

The representation of all suffixes of a string in form of a suffix tree allows to run many different algorithms that solve common search problems in strings in adequate time [McC76]. Some example questions that can easily be answered with the suffix tree built for the string X are:

- Is string W a substring of X?

- Find all occurrences of a substring S in X.
- Find all maximal matches of substrings in X.

The last question is the one that is relevant for finding clones in software. Assuming the whole source code is parsed into one token string, the search for maximal matches of substrings reveals code fragments that are equal and have therefore most probably been copied.

2.4 Clone Pairs and Suffix Trees

This section describes the relation between suffix trees and clone pairs. This relation is used by *Baker's* algorithm for extracting maximal matches from a suffix tree and becomes relevant when the structure of the suffix tree is to be modified.

Each internal node of a suffix tree represents a sequence of characters or tokens that appears more than once inside the string from which the suffix tree was built. As the sequence has at least one identical copy, it is a clone. The token sequence of the clone is identical to the path of the internal node. How often the sequence appears in the string is determined by the number of leaves that can be reached from the internal node, because every leaf represents a different suffix of the string.

Among these clones, every pair that can be formed is a clone pair as defined in Section 2.2. However, many of these pairs are less interesting, because they are not maximal and covered by other pairs. According to *Baker*,

> *"A match is maximal if it is neither left-extensible nor right-extensible [. . .]"* [Bak93].

A clone pair $cp = ((file_1, start_1, end_1), (file_2, start_2, end_2), type)$ is said to be *right-extensible* if the token in $file_1$ at position $end_1 + 1$ equals the token in $file_2$ at position $end_2 + 1$. If that is the case, the match is not maximal, because both fragments can be expanded by one token to the right. If either end_1 or end_2 is the last token of the respective file, the match is not right-extensible, because at least one token is undefined.

A clone pair $cp = ((file_1, start_1, end_1), (file_2, start_2, end_2), type)$ is said to be *left-extensible* if the token in $file_1$ at position $start_1 - 1$ equals the token in $file_2$ at position $start_2 - 1$. This is also not recognized as a maximal match. Analogous to right-extensibility, if $start_1$ or $start_2$ denotes the first token of the respective file, the clone pair is not said to be left-extensible.

There is however one problem concerning this definition. It does not consider the case where one fragment is left-extensible and the other fragment is right-extensible and both fragments still contain the same sequence of tokens. Such situations appear in conjunction with self-similar fragments. This results in clone pairs being reported as maximal although both fragments are extensible. Because there is no satisfactory solution to this problem yet, this thesis will use the definition as it has been presented.

Concerning the extraction of clone pairs from the suffix tree, right-extensibility does not need to be explicitly checked if only fragments are combined that stem from leaves which are reached by different edges from an internal node. According to the definition of suffix trees, no pair of outgoing edges from that node can share the first character (or token) of their label. As this token is the first to the right and different for both fragments, the clone pair cannot be right-extensible. If however clone pairs were formed by leaves being reached from the same outgoing edge, the respective fragments must by definition be right-extensible. At least by the tokens of the outgoing edge which they share.

Left-extensibility can unfortunately not directly be read from the suffix tree and must be tested upon combining two fragments. If the resulting clone pair is left-extensible it is discarded, otherwise it is reported as being maximal.

Having found two leaf nodes n_1 and n_2 that share a sequence of tokens which is neither left-extensible nor right-extensible, a clone pair can be built as follows. The value of $file_1$ equals the file information contained in the label of the parent edge of n_1. The indices $start_1$ and end_1 can be obtained by considering the indices of n_1's parent edge and the path length of node n_1. The same applies to the second fragment using the node n_2. The clone pair's type is determined later in a post processing step. Important is, that all values can be determined in constant time.

Summing up, fragments relate to leaf nodes and clone pairs to internal nodes of the suffix tree. Indices for these can be calculated in constant time. Figure 6 shows a maximal match in a sample suffix tree.

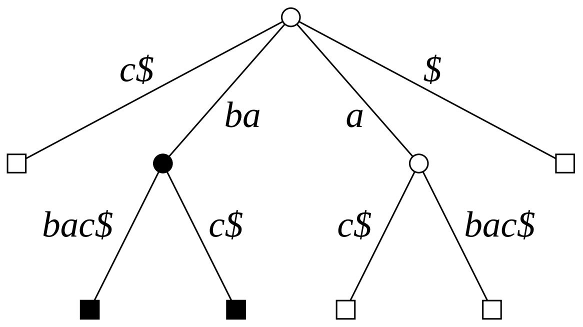


Figure 6 – Maximal match in the suffix tree for the string *babac*\$. The black external nodes represent the fragments of the clone pair. The path from the root to the black internal node shows the cloned sequence of characters being *ba*. Note, that the path with label *a* of the second internal node is not a maximal match, because *a* is left-extensible.

2.5 Clone Detection Using *clones*

Different approaches to clone detection have been briefly outlined in Section 1.2. This section explains the process of token-based clone detection as implemented in the tool *clones* from the project *Bauhaus*. Due to the underlying task of this thesis, the usage of *clones* and the adaptation of *clones* mechanisms to multi-revision clone detection is compulsory.

The tool *clones* runs five major phases to detect clones in a given program. First, all source files are parsed into one large token string, then the suffix tree is built for that string. Within the suffix tree, *Baker's* algorithm [Bak97] is used for finding maximal matches which are filtered afterwards in order to discard uninteresting clone pairs. The last step consists of outputting the resulting set of clone pairs in the desired format.

2.5.1 Tokenizing

This is the first step in token-based clone detection and common to every compiler and every program that analyzes source code. All files that belong to the system in which clones are to be searched are collected and scanned by a lexer into a string of tokens. The token strings of all files are concatenated together to form a single string. This allows building a single suffix tree which contains the information about all suffixes of all files. If a single suffix tree was used for each file, no clone pairs could be found where the fragments stem from different files. A single tree for all files ensures that clone pairs can be found across files. Before concatenation, a unique file terminator token is appended to the token string of each file. This ensures, that no fragments that cross file boundaries are part of clone pairs. The concatenated string is saved in a single token table that maps an index to information about the token at that position.

2.5.2 Suffix Tree Construction

After creating the token string of all tokens from the source code, the suffix tree for this string is built. Depending on the application, two construction methods for the suffix tree are available. The first construction algorithm is an implementation of *Ukkonen's* algorithm [Ukk95], which directly builds the suffix tree for the token string. Although the algorithm has the benefit of being on-line, this is not important for *clones*, because the complete string for which the suffix tree is built is available before starting the suffix tree construction.

Another suffix tree constructor is available which implements the algorithm presented by *Baker* [Bak97]. The token string is first converted into a parameterized token string. Based on the parameterized token string, the constructor builds a parameterized suffix tree. This constructor is used whenever parameterized clone detection is requested, because the *Ukkonen* constructor is not able to build a parameterized suffix tree.

2.5.3 Extraction of Longest Matches (Baker)

In the third step, *Baker's* algorithm *pdup* [Bak97] for extracting longest matches from the suffix tree is used to find potential clone pairs. To reduce the number of clone pairs that need to be processed in the following phases, some potential clone pairs can already be discarded. The most important criteria for discarding clones is the minimum length they need to have. The algorithm is already aware of this criteria and drops clone pairs which are not long enough to be reported. The result is a list of clone pairs, that all conform to the minimum length.

2.5.4 Filtering

Though a lot of clone pairs have already been dropped during the extraction of longest matches, the list of pairs most probably contains a lot of pairs that are of no interest (i.e. overlapping pairs or pairs which span more than one syntactic unit as shown in Figure 7). To improve the result, several filtering phases are run.

The first phase determines the type of each clone pair. Due to the nature of *pdup*, only type-1 and type-2 clone pairs are extracted from the suffix tree. For each clone pair, the fragment's identifier tokens are checked for equality. If a discrepancy is found, the pair is of type 2, otherwise of type 1.

Other filters include cutting clone pairs down to syntactic units, merging type-1 or type-2 clone pairs into type-3 pairs or removing overlapping clone pairs. Depending on the filters that are applied, this step is quite time intensive. It is not unusual, that a filter has a worst-case quadratic time complexity as every clone pair needs to be compared to every other clone pair.

2.5.5 Output

The last step consists of formatting the clone pairs and outputting them in the desired format. The information about the location of fragments needs to be converted from token indices into lines, to make the result readable for the user. Depending on the required format, each clone pair is emitted with the location of the two fragments and its type as well as its length.

```
    a = a + 1;
  }
}

void do_it()
{
```

(a) $fragment_A$

```
    b = b + 1;
  }
}

void undo_it()
{
```

(b) $fragment_B$

Figure 7 – A clone pair that is not very helpful as the token sequence of the fragments spans more than one syntactic unit.

3 Incremental Clone Detection

This section describes the ***I**ncremental **D**etection **A**lgorithm IDA* and concepts relevant to it. The motivation for *IDA* is to have an algorithm that analyzes multiple revisions of a program faster than the separate application of the tool *clones* on each revision. The motivation is based on the assumption that only a comparatively small amount of files change per revision, causing a lot of work to be done redundantly when rerunning every part of *clones*. Therefore, internal results are not discarded, but reused and modified according to the files that changed for the respective revision.

The information which files have changed between any two consecutive revisions is given to *IDA* together with the source code of each revision that is to be analyzed. *IDA* itself runs several phases for every revision, which consist of *preparations*, *processing changes* and a *post processing* stage.

To accelerate the detection of multiple revisions, the analysis of $revision_i$ must consider and reuse intermediate results of $revision_{i-1}$ as much as possible. The intermediate results which can be reused are the data that are created during the clone detection process. A lot of this data are not directly part of the token-based clone detection process. Though being also reused as far as possible, they are not mentioned here due to their technical nature.

Answering Question 1, the data structures that are important for the clone detection process and which can potentially be reused are the token table, the suffix tree and the set of clone pairs. The time needed to create these takes a considerable amount of time of the whole detection process. For this reason, they are not discarded after the analysis of a revision, but kept in memory and reused for the next revision. However, some effort must be spent to make these data structures suitable for multi-revision detection.

Sections 3.1 and 3.2 answer Question 2 by explaining in which way *clones'* data structures must be modified in order to be reused. Section 3.3 shows how the incremental algorithm integrates into the analysis of multiple revisions. Sections 3.4 to 3.6 explain the different phases of which *IDA* consists.

3.1 Multiple Token Tables

The first data structure which is to be reused is the token table. As files are added and deleted from revision to revision, old tokens that are not needed anymore can be discarded and new tokens have to be read for new files. If a single table is used, problems arise when files change and the token tables for the respective files need to be updated to conform to the new versions.

Assuming a file is deleted, its tokens have to be removed at some point from the token table, because otherwise the algorithm will sooner or later run into memory shortage. Recalling that the edges of the suffix tree are labeled by the start and end index of a substring, one has to ensure, that no label of the suffix tree references the part of the token table where tokens were discarded. This would require checking the complete suffix tree for dangling indices upon deleting a file. Even more critical, new indices that denote the same substring have to be found for every edge with invalid indices.

Apart from invalid indices, there is the problem of growing holes inside the token table, caused by files that have been deleted. One could try to fit new files into these spaces and fill the holes, but whatever token table management is used, the management overhead will most likely be intolerable. More complex methods, that remap indices to valid locations upon requesting a token are of no use, because accessing a token from the token table is the most common action in the implementation and should therefore be as fast as possible. Any yet so small delay in accessing a token will add up to a significant amount.

These issues are the reason for *IDA* to not hold a single token table, but rather have one token table for every file. When a file is deleted, the token table for the respective file can just be dropped after the suffix tree has been updated. When a file is added, a new token table is created. The location of a token must therefore be extended to a tuple $(file, index)$ instead of just having a single index. *file* selects the token table for the file in which the token is contained, and *index* denotes the position of the token inside that file. Section 3.2.2 describes how dangling references caused by the deletion of files are avoided.

3.2 Generalized Suffix Tree

In addition to changing the token tables when files have changed, suffixes of these files also have to be added and deleted from the suffix tree. This causes the structure of the tree to change in order to conform to the new token sequences of the changed files. Assuming concatenated token strings and a suffix tree for the complete string as practiced in *clones*, every external edge's label spans all tokens to the very end of the token string including the tokens of all files following the edge's file. If the token string becomes shorter (or longer), the label's end index of every external edge would need to be changed. Many parts of the suffix tree have to be modified that are in no way related to the file which has changed. It is desirable to only modify the edges related to the changed file and leave all other edges untouched.

To solve this problem, *IDA* uses a *generalized suffix tree* [GLS92] that represents suffixes of multiple strings instead of just a single string. The advantage of a generalized suffix tree is, that algorithms exist to efficiently insert or remove a string from the underlying set of strings and update the tree accordingly by only modifying the edges relevant for the respective file.

A generalized suffix tree represents all suffixes of all strings in a set of strings $\Delta = \{X_1, X_2, \ldots, X_n\}$. It can be seen as the superimposition of the individual suffix trees of the strings in Δ. Whenever parts of edge labels are equal, they can be unified into a single edge, now serving for two or more strings from Δ. However, the tuples that label the edges must be expanded, because more than one string contributes to labeling the edges. The tuple (i, j) which referred to start and end position inside the string is extended to a triple also giving information to which string the indices i and j refer. An edge label is now a triple (X, i, j), where i and j denote the start and end position of a substring in X. A sample generalized suffix tree for the set of strings $\Delta = \{abc\$_1, bab\$_2, bac\$_3\}$ is given in Figure 8. It should be kept in mind, that every string X_i in Δ has a unique endmarker $\$_i$, which does not match any other character or endmarker.

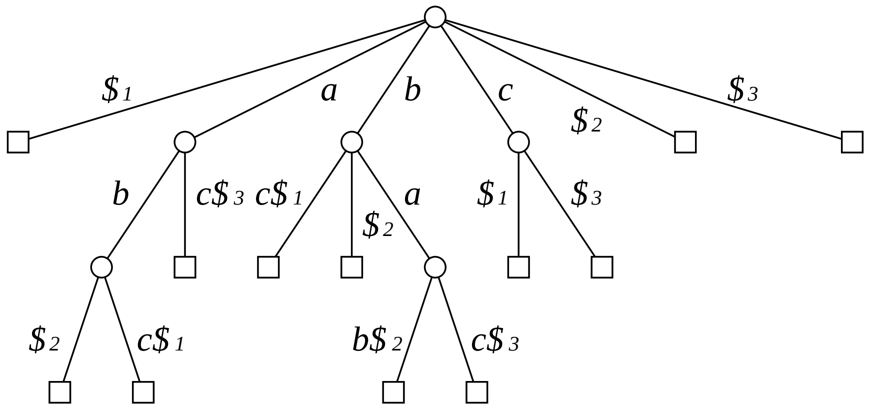
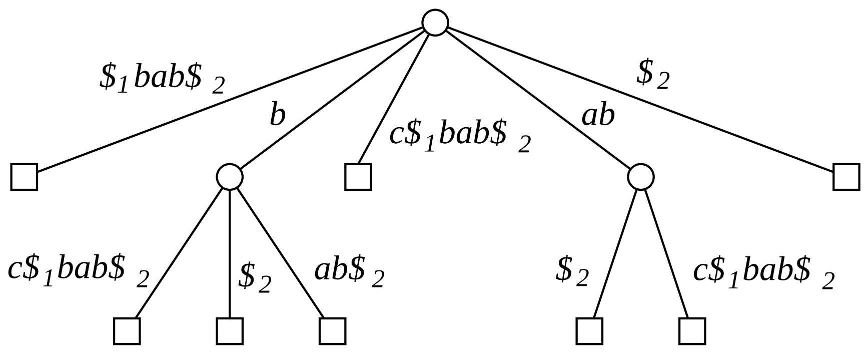


Figure 8 – Generalized suffix tree for the set of strings $\Delta = \{abc\$_1, bab\$_2, bac\$_3\}$.

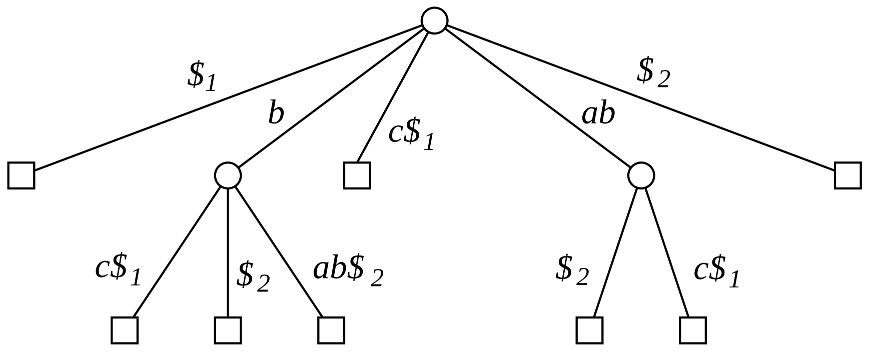
It is worth noting, that the generalized suffix tree for $\Delta = \{X_1, X_2, \ldots, X_n\}$ is isomorphic to the suffix tree which can be constructed from the concatenated string $X_1X_2 \ldots X_n$.

Furthermore an important property of generalized suffix trees is, that suffix links always point to nodes that represent a suffix of the same string. By following the suffix links until reaching the root node, all suffixes of a given string can be retrieved and no nodes representing suffixes from other strings are traversed.

Figure 9 compares a conventional suffix tree for the string $abc\$_1bab\$_2$ to a generalized suffix tree for $\Delta = \{abc\$_1, bab\$_2\}$. They differ in the labels of the external edges. If the substring $bab\$_2$ was removed from the conventional suffix tree, every external edge would have to be relabeled. In the generalized suffix tree, only those edges whose labels end on $\$_2$ would need to be modified.



(a) Conventional suffix tree.



(b) Generalized suffix tree.

Figure 9 – Comparison of a conventional suffix tree for the string $abc\$_1bab\$_2$ to a generalized suffix tree with $\Delta = \{abc\$_1, bab\$_2\}$.

The remaining question is, which syntactic units of the analyzed programming language are to be represented by the strings in Δ. Considering the relation between the suffix tree and clone pairs, one can notice, that the maximal length of a clone pair is limited by the length of a string in Δ. If the strings in Δ would represent single statements of the program, no clone pairs could be found which are longer than a single statement, which is undesirable. Making strings represent syntactic blocks like functions would require understanding the syntactic structure of the program. This is however not given, as token-based clone detection does not give any meaning to the sequence of tokens. Furthermore, clone pairs that span more than one syntactic unit might be of interest but would not be detected.

The choice was made to make each string in Δ represent the token string of a single file. This allows removing or adding a single string to Δ for a single changed file. Furthermore, fragments can not cross file boundaries due to the length limitation mentioned above.

One extension is made to the generalized suffix tree that is specific for *IDA*. Every external node of the suffix tree stores references to the clone pairs of which a fragment relates to this node. On the other hand, every clone pair maintains links to the two nodes to which its fragments correspond. These bi-directional links are exploited whenever the structure of the suffix tree is modified and *IDA* requests all fragments relating to an external node. When external nodes are removed from the suffix tree due to the corresponding suffixes being deleted, fast access to the clone pairs is required, because these need to be changed. Note, that each clone pair is linked to exactly two nodes, whereas each node can be linked to an arbitrary number of clone pairs. Figure 10 shows how pairs and external nodes are linked.

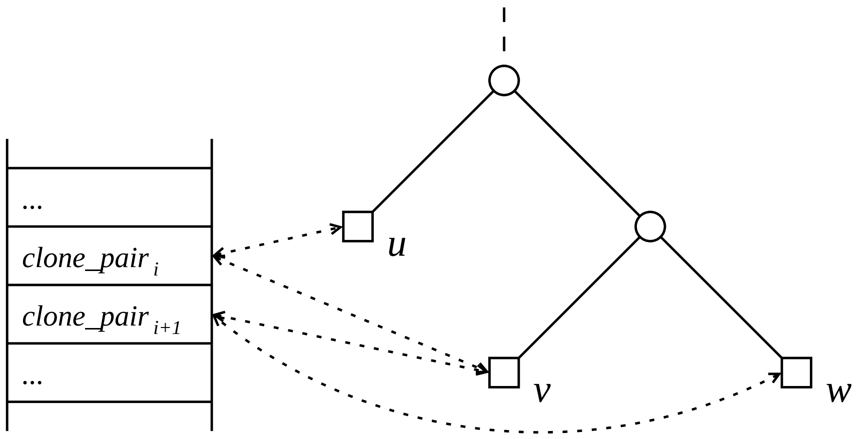


Figure 10 – Schematic view of bi-directional links (dotted arrows) between external nodes and the fragments of clone pairs. *clone_pair*$_i$ has one fragment related to node u and the other fragment related to node v. Likewise, *clone_pair*$_{i+1}$*'s* fragments relate to nodes v and w.

3.2.1 Tree Construction

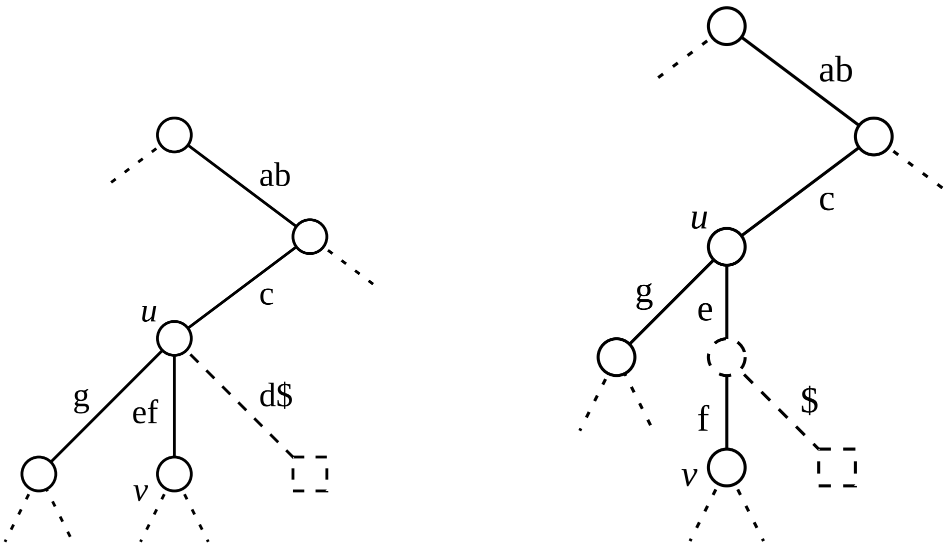
Unfortunately, the construction methods for suffix trees that are implemented in *clones*, are not flexible enough to allow for fast insertion and deletion of strings into a generalized suffix tree. *IDA* implements unparameterized clone detection, because the impact of adding or removing a parameterized string from a parameterized generalized suffix tree has not been evaluated within this thesis. This makes *Baker's* constructor for parameterized suffix trees unusable for *IDA*. The constructor that implements *Ukkonen's* algorithm is also not usable, as it processes a single string online from its first to its last character. The benefit of being on-line is not required in our application, as parts of the string do change after they have been processed once.

Instead, *IDA*'s tree construction is based on *McCreight's* algorithm [McC76]. *McCreight* was the first to give a comparatively simple algorithm for constructing a suffix tree in linear time. In contrast to *Ukkonen's* algorithm, suffixes are added from longest to shortest to the tree. While constructing the tree, suffix links are created and exploited in later iterations. The drawback of *McCreight's* algorithm is, that it operates "backwards" starting with the longest suffix. This means the whole string has to be known upon starting the algorithm. Nonetheless, this method is used by *IDA* to construct and modify the suffix tree. There is no problem in starting with the longest suffix, because the program's source code is completely available upon starting the clone detection. Furthermore, *McCreight's* algorithm can easily be applied to generalized suffix trees.

However, *IDA* does not follow *McCreight's* original paper, but a simplified version which is described by *Amir et al.* [AFG+93]. *Amir et al.* present two procedures. One of them inserts a suffix of a string into the (generalized) suffix tree and the other one accordingly deletes a suffix from the tree. Assuming that all suffixes of a string are added or deleted (and not a single one on its own), both procedures take constant time. The assumption is valid for *IDA*, because whenever a file is changed, all its suffixes are added or deleted. A string of length m, which can be seen as a file with m tokens, has m suffixes and can therefore be added or deleted from the suffix tree and its underlying set of strings Δ in linear time $O(m)$. Like with other constructors, the generalized suffix tree is augmented with suffix links. A brief explanation of both procedures is given. A detailed description and a proof why they require linear time can be found in [AFG+93].

Inserting a Suffix: Inserting a suffix of a string into the generalized suffix tree is done by creating a new external node representing that suffix together with a new external edge. The challenge is to identify the location where the new edge is to be appended. This is done by following the edges away from the root as long as the tokens represented by the edge labels correspond to the tokens of the suffix. There is only one such path, because the edge labels of sibling edges all start with a different token. When a discrepancy is found between the token referred to by an edge label and the corresponding token of the suffix, the location for inserting the new external edge and node has been found.

If the discrepancy was caused by the first token of an edge, the new external edge and its end node can be appended to the parent node of that edge. Otherwise, the existing edge needs to be *split* by inserting a new internal node to which the new external edge must be appended. The new external edge label refers to the remaining tokens of the suffix for which no match has been found. Figure 11 shows both situations.



(a) Appending to an existing node. **(b)** Splitting an edge.

Figure 11 – Appending a new external node and edge to the suffix tree. Assuming the suffix to be added is *abcd*\$, the new edge and node (dashed) can be appended to node u (a). If the suffix is *abce*\$, the edge from node u to v needs to be split. The new edge is appended to the newly created internal node (b).

During the process of inserting suffixes, suffix links are created. By using these links, the search does not need to be started from the root node for each remaining suffix. Instead, an internal nodes serves as the starting point for the search to avoid redundant calculations and ensure linear time consumption for inserting all suffixes of a given string.

Deleting a Suffix: Deleting a suffix is similar to adding one. It implies deleting an existing external node and its parent edge. Finding the node that is to be deleted works analogous to finding the location where a new node is to be appended. Starting from the root, edges are traversed as long as the tokens referenced by their labels equal the tokens from the suffix that is to be deleted. The search ends at the external node that represents the suffix. Again, two situations can arise. If the parent node of the external node is the start node of more than two edges, the external node and its external parent edge can be removed. If the external edge has only one sibling, the sibling and the parent edge of its start node have to be merged after removing the external edge. This is required, because no node except for the root can be the start node of only one edge.

Merging two edges is the opposite of splitting them. Coming back to Figure 11 and assuming the dashed parts to be deleted, subfigure (a) requires no further modification of the tree. When the dashed edge is removed from subfigure (b), its start node remains the start node of only one other edge. This requires merging the edges from u to the start node and from the start node to v.

IDA always deletes suffixes from longest to shortest. If the external node representing the longest suffix has been found, any other node representing a shorter suffix of the same string can be retrieved by following the suffix links. This means, every following suffix can be deleted in constant time.

3.2.2 Updating Labels

There is a serious problem in modifying generalized suffix trees after their initial construction. One should recall that every edge is labeled by a triple consisting of *file*, *start* and *end* referring to a substring of tokens. Considering this, deleting files might result in edges, whose labels point to files which are not existent any more. A solution to this problem is described by *Ferragina et al.* [FGM97]. They make the following statement about labels:

> *"[...] a label (X, i, j) is consistent if and only if it refers to a string currently in Δ."* [FGM97]

Using this definition, every label of a generalized suffix tree must be consistent at any point of time. *Ferragina et al.* identify three basic operations that modify the structure of the suffix tree.

- Inserting a new external node and edge into the tree. The start node of the new edge already existed before the insertion. This operation appears in conjunction with the insertion of a string into Δ (see Figure 11a).
- Inserting a new external node and edge into the tree. The start node of the new edge did not exist before the insertion, but was created by splitting an existing edge. This operation is also related to the insertion of a string into Δ (see Figure 11b).
- Deleting an external node and edge due to the removal of a string from Δ. If the start node of the edge is the start node of only one other edge, the other edge and the nodes parent edge have to be merged.

The solution presented to keep suffix tree labels consistent under these three operations requires maintaining bi-directional links between edges and nodes. Using these links, the labels that need to be modified when one of these operations is carried out, can be identified in constant time. Relabeling an edge implies changing the file to which the label refers. This most likely leads to changing the start and end index of the label, because the actual sequence of tokens which is referred to by the label must not change.

Ferragina et al. introduce three conditions which must hold at any point of time in order to ensure, that relabeling an edge can happen in constant time. Satisfying these conditions might require relabeling edges although nothing is deleted from the tree. As the tree can be kept consistent in constant time for an edge, keeping the tree consistent when removing a string from Δ requires time linear to the length of that string. More details on how exactly links are built and when edges need to be relabeled can be found in [FGM97].

3.3 Integrating *IDA*

After Sections 3.1 and 3.2 answered Question 2, the remaining sections present the incremental clone detection algorithm and answer Question 3.

Before the individual phases of *IDA* are described, a general view is given of how *IDA* integrates into multi-revision clone detection. Recalling Figure 1, one can observe that the same processing steps are done for each revision. The analysis of each revision is independent of any other revision. Figure 12 shows how *IDA* integrates into the analysis of two consecutive revisions.

The intermediate results from $revision_i$ are the tokens stored in token tables, the suffix tree and the clone pairs. These are given to *IDA* as input together

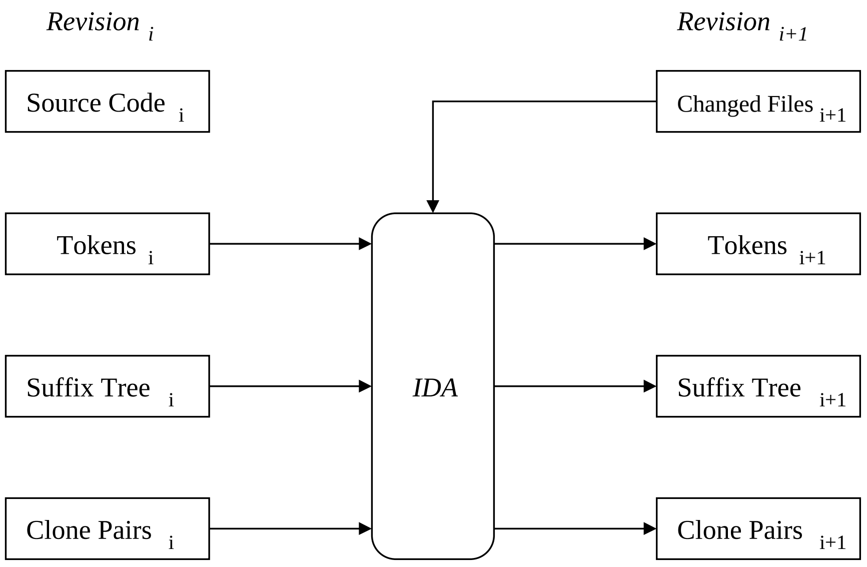


Figure 12 – Integration of *IDA* into the analysis of two consecutive revisions. Arrows indicate input and output of *IDA*.

with the files that changed from $revision_i$ to $revision_{i+1}$. Depending on the changes of the source files, the data structures are modified in order to conform to the source code of $revision_{i+1}$. Again, they are kept in memory in order to be reused for the next revision.

3.4 Preparations

After describing how *IDA* integrates into clone detection, the following sections will explain its internal processing. Like with most other algorithms, certain preparations have to be made before the main part of *IDA* can execute. A distinction has to be made between preparations that have to be done only once and others which have to be done for every new revision which is analyzed.

Initial preparations are those, that have to be done only once for every execution of *IDA*. Analogous to most other programs, *IDA* needs to set up various data structures which are used during the clone detection process. This is not further described here, because this is purely technical and not part of the algorithm itself.

IDA starts by reading the directory that contains the individual revisions. It assumes the first directory in alphanumerical order to be the first revision of the program which is analyzed. As there are no intermediate results to be used, everything has to be created from scratch. *IDA* processes the first revision in the conventional way as implemented in the tool *clones*. A description of which steps are performed is given in section 2.5.

In contrast to preparations that have to be done only once, there are certain things that must be done every time before a new revision is analyzed. These mainly consist of technical issues like resetting certain data structures.

3.5 Processing Changes

After doing the preparations, *IDA* is ready to process all the files that have changed. The overall procedure is rather simple and consists of going through the list of changed files. Depending on the type of the change, the token tables, the suffix tree and the set of clone pairs are modified. Adding or deleting a file are the simple cases, processing a modified file is more complex. The following sections describe how the different types of changes are processed.

3.5.1 Processing a New File

If a new file was added in the current revision, *IDA* constructs a new token table for that file and calls the lexer in order to fill the table with the file's tokens. After that, the file's suffixes are added to the suffix tree from longest to shortest. For every suffix of the file, one new external edge is created representing that suffix. This might require splitting an existing edge and relabeling edges to allow keeping edge labels consistent. For details on how to add or delete suffixes, edges and nodes from the suffix tree see [AFG+93] and [McC76].

Every edge in the suffix tree carries a tag that indicates the status of the edge. The status can be either *new* if the edge has been added to the suffix tree, *reused* if it was reused (see Section 3.5.3) or *none* if the edge has not been modified. Upon adding a file, all new edges are tagged as being new, which is important when rerunning *Baker's* algorithm to retrieve clone pairs from the suffix tree (3.6.1). If required, other edges are relabeled as described in section 3.2.2. The generalized suffix tree for $\Delta = \{abc\$_1, bab\$_2, bac\$_3\}$ after the insertion of $bac\$_3$ is shown in Figure 13. Dashed edges are the ones tagged as *new*.

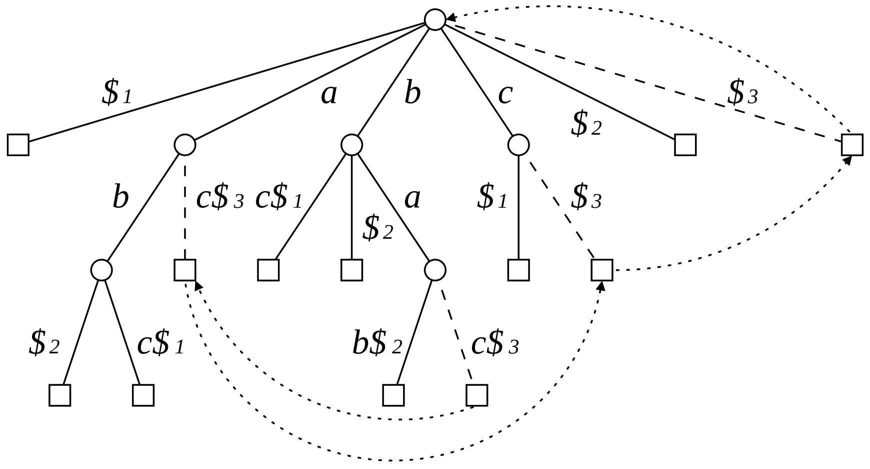


Figure 13 – Generalized suffix tree for $\Delta = \{abc\$_1, bab\$_2, bac\$_3\}$. Dashed edges and their end nodes relate to the string $bac\$_3$. Dotted arrows indicate suffix links connecting the external nodes representing the suffixes of $bac\$_3$. Note, that not all suffix links are drawn to limit the figures complexity.

3.5.2 Processing a Deleted File

Like addition, deletion of a file is rather simple. Starting with the longest suffix, the external edge and its end node that represents that suffix are searched and deleted. Assuming the string $bac\$_3$ is deleted from the tree in Figure 13, the end node representing the longest suffix is the one whose path is $bac\$_3$. Deleting an edge might require merging or relabeling other edges as described in 3.2.2. The nice thing about deletion is, that starting from the end node of the deleted edge, suffix links can be followed to collect all edges which represent suffixes from the file that is deleted. This can also be observed in Figure 13. The suffix links that can be followed are drawn as dotted arrows. As every node carries a reference to its parent edge, the edges which have to be removed can easily be obtained.

In addition, all clone pairs from which at least one fragment is contained in the deleted file, are removed because they do not exist any longer. Finally, the token table for the file that is deleted can be disposed, as the tokens are not needed anymore.

3.5.3 Processing a Modified File

In comparison to addition or deletion, modifying a file is by far more complex. Basically, a modification of a file is processed by removing the old version and adding the new version of the file to the suffix tree. There are however some particularities and some thought should be given of what happens to the suffix tree upon modification of a file. Assuming the content of the file does not change at all, every external edge representing a suffix of the old version of the file would be deleted and the same edge inserted afterwards for the new version of the file. The only difference would be that the labels of the external edges now refer to the new, instead of to the old version of the file. Though inserting and deleting edges is relatively fast, a lot of work has to be done which is actually not needed. To overcome this problem, the concept of *reusing* edges is introduced.

Starting like addition, a new token table is created for the new version of the file and the tokens of the file read and inserted into the table. Starting with the longest, suffixes are inserted into the suffix tree. However, there is a big difference in comparison to the insertion of a new file. Whenever an external edge is to be inserted into the suffix tree, its potential siblings are compared to the edge. If there is any external sibling edge label that references the same sequence of tokens as the new edge would do (except the file terminator token which is different for every file) and whose label points to the old version of the file, it can be reused. This prevents any structural modification of the suffix tree. The edges label is made to reference the new version of the file and the edge is tagged as reused. This can save huge amounts of calculations which would otherwise be needed in order to modify the suffix tree. Figure 14 illustrates a situation where an edge can be reused.

If a reusable edge is found for a particular suffix, there needs to exist a reusable edge for any remaining suffix which is to be inserted into the tree. Once more, the suffix links can be traversed starting from the end node of the reusable edge until the root node is found. Every parent edge of the nodes that are traversed can be reused and no more modification has to be done to the tree except for changing file references in edge labels. For example, Table 2 shows the old and new version of the tokens of a file where the second token with index 1 has been changed from `a` to `b`. The suffixes $[i, 3]$ with $2 \leq i \leq 3$ do not require a structural modification of the suffix tree, because these suffixes already existed in the old version of the file. Only the labels of the edges have to be adapted.

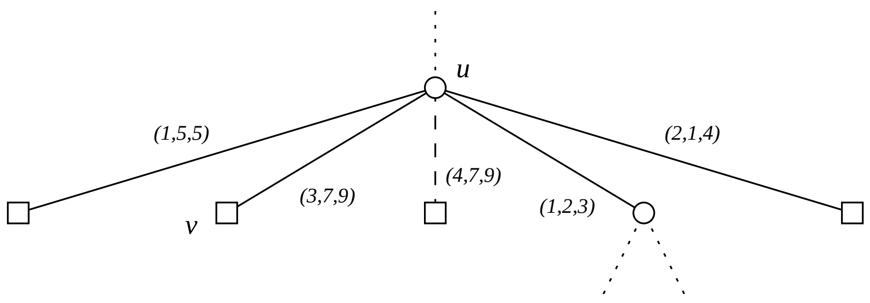


Figure 14 – A situation where an edge can be reused. The index of the old version of the file is 3, the new version has index 4. If the new external edge with label $(4, 7, 9)$ (dashed) is to be appended to node u, the edge from u to v can be reused by making the label reference file 4 instead of 3. This assumes, that the tokens 7 and 8 in the old version of the file equal tokens 7 and 8 in the new version and that token at index 9 is a file terminator token, which is the case as every external edge needs to end in a file terminator token.

Index i	0	1	2	3
Old version	`b`	`a`	`c`	$\$_3$
New version	`b`	`b`	`c`	$\$_3$

Table 2 – The old and new version of a token string. The token at index 1 has been changed from `a` to `b`.

One might already have recognized, that the time that can be saved heavily depends on the location of the tokens which have changed from the old to the new version. Due to the nature of a suffix tree, the closer a change is to the end of the string, the less there is to gain from reusing edges. The token at position i belongs to $i + 1$ suffixes. If that token is changed, $m - (i + 1)$ edges can be reused where m denotes the length of the token string. The closer i gets to m, the less there is to reuse.

After inserting the new version of the file, the suffixes of the old version of the file need to be removed from the tree by deleting the corresponding external edges. The edges are once more traversed by following the suffix links that connect the end node of these edges. Whenever an edge is removed from the tree, the consistency of the tree must be preserved. This is done by relabeling edges as described in section 3.2.2. Furthermore, deleting an edge might require merging two edges if the deleted edge had only one sibling. For any edge deleted this way, the clone pairs related to this edge are removed as the fragment represented by the edge's end node does not exist any longer. As soon as an edge tagged as reused is found, the procedure can terminate because any following edge has also been reused as described earlier.

Figure 15 shows the suffix tree for $\Delta = \{abc\$_1, bab\$_2, bbc\$_3\}$ after changing file $bac\$_3$ to $bbc\$_3$. Two edges can be reused, because the respective suffixes did not change. Note how the suffix links connect all nodes representing suffixes of the string $bbc\$_3$ and allow for easy traversal of these nodes.

3.6 Post Processing

When all changed files have been processed, the token tables and the suffix tree conform with the new revision of the program analyzed. However, the set of clone pairs does not. To make the clone pairs comply with the new revision, two post processing steps are required.

3.6.1 Searching New Clones

When new files have been added or any file has been modified, the suffix tree contains edges tagged as new. As the end nodes of these edges might relate to fragments which form new clone pairs, these pairs have to be retrieved. It is important to note that new clone pairs cannot only be formed among new edges and their end nodes. Any new end node might also form a clone pair with an end node of an edge that is not tagged as new. Therefore, the huge

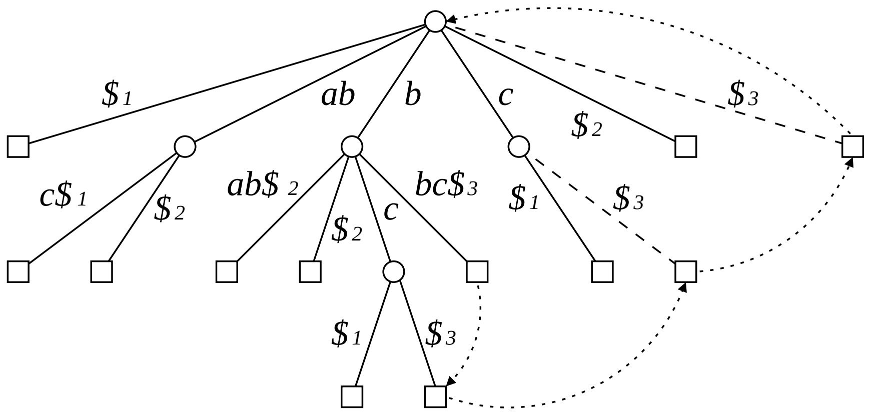


Figure 15 – Generalized suffix tree for $\Delta = \{abc\$_1, bab\$_2, bbc\$_3\}$ after file $bac\$_3$ has been modified to $bbc\$_3$. Dashed edges show the edges that were reused, because the suffixes $c\$_3$ and $\$_3$, represented by the edge's end nodes, did not change. Dotted arrows indicate suffix links connecting the external nodes representing the suffixes of $bbc\$_3$. Note, that not all suffix links are drawn to limit the figures complexity.

amount of possible combinations inhibits a specialized search for potential clone partners for every new edge.

Instead, new clone pairs are retrieved by rerunning *Baker's* algorithm on the whole tree. But in comparison to its original version, the algorithm now only reports clone pairs where at least one of the fragments relates to the end node of an external edge tagged as new. This modification ensures, that no clone pairs are reported which are not related to at least one new node and which therefore have been extracted before. The result is a set of potentially new clone pairs of which at least one fragment did not exist previously.

3.6.2 Filtering

Analogous to processing the first revision, the resulting set of potentially new clone pairs is filtered to match the criteria specified by the user. Pairs that do not conform to the filter criteria are dropped and not considered any more.

Apart from filtering the new clone pairs, the set of existing clone pairs has to be reviewed. All these pairs have to be checked for left-extensibility once more, because changing files and modifying the suffix tree makes the initial left-extensibility check invalid. Any clone pair might have become left-extensible due to a changed token sequence in changed files. If a pair is found to be left-extensible, it is deleted, because of not being maximal any longer.

The new clone pairs which have not been discarded by the filter are integrated into the set of existing clone pairs and the result represents all clone pairs which exist in the revision that is currently analyzed.

Finally, a number of clean-up actions are performed, which are not described here in detail due to their technical nature. It is worth noting, that all data structures now completely conform to the current revision which has just been analyzed. The token tables, the suffix tree and the set of clone pairs are exactly complying to that revision and no relics from previous revisions exist. After everything is done, *IDA* processes the next revision or terminates if no more revisions are to be analyzed.

4 Tracing

The evolution of clones has an important role in understanding and estimating the impact of clones on a software system [HK08]. During the last years, several approaches have been presented to analyze clones over more than one revision of a program [ACPM01, AVMP02, DER07, KSNM05, KN05, Kri07]. Applications range from simple quantitative measurements to more complex questions directed at the relation between clone pairs or classes, i.e. consistent or inconsistent changes of fragments within a class or a pair. A short summary is given of how clones are traced in different works.

Antoniol et al. presented a method to analyze the evolution of clones quantitatively over multiple revisions of a program [ACPM01, AVMP02]. The average amount of clones per function is measured for different revisions of the program. Based on the results, a model is created to predict how the clones further evolve. Clones of function granularity are detected by comparing functions according to their metrics. However, this approach just considered the total amount of clones and does not create a mapping between clones of different revisions.

Kim et al. studied clone genealogies in 2005 [KSNM05, KN05]. By tracing clone classes over multiple revisions, a genealogy for each of those classes can be constructed. The matching between clones of different revisions is done by a heuristic function which measures the location overlapping of both fragments. The more the locations of a cloned fragment in one revision overlaps with a cloned fragment from another revision, the more likely both fragments are the same clone. Furthermore, *Kim et al.* define different patterns to describe the evolution of clone classes.

A study of consistent and inconsistent changes to clones was done by *Krinke* [Kri07]. A format for describing the location and extent of a code change from one revision to the next is presented. All changes in the source text of one revision to the next are expressed in that format. Each of these changes is analyzed, whether its location is embedded or at least overlaps with the code fragment of a clone. Based on the outcome, information about whether clone changes are consistent or inconsistent can be gained.

Duala-Ekoko and *Robillard* introduced *Clone Region Descriptors (CRDs)* [DER07]. A CRD is an abstract representation of the location of a clone in the source code. Looking at the abstract syntax tree of any revision of the program, a search routine tries to retrieve the clone based on the CRD. The CRD stores a clone's location not by absolute information about source code lines which the clone spans, but rather by naming syntactic structures which contain the clone. This allows finding clones although they might have moved due to the insertion or deletion of source code lines from the respective file.

Balint et al. presented a visualization method for the evolution of clone classes [BMG06]. Their *Clone Evolution View* shows the timeline for a single class and enriches it with information about the date and authors that changed source code lines of the clones. They retrieve data about the changes made to the clones from the version control system *CVS*[4].

Running the incremental clone detection algorithm as described in the previous section still yields independent sets of clone pairs. There is no gain concerning the mapping between clone pairs of individual revisions in comparison to the separate application of *clones* to each revision. One way to obtain the mapping is matching clone pairs of different revisions after the creation of the sets. If a match of two sufficiently similar clone pairs is found, it is assumed that these pairs are essentially the same. However, this approach can be very costly for large sets of clone pairs due to the quadratic nature of the matching phase.

The integration of tracing clone pairs across revisions into *IDA* is based on the assumption, that changes of clone pairs can be derived from the modification of the suffix tree. This reduces the amount of clone pairs that need to be matched afterwards.

In addition to mapping clone pairs, the change that happened to each pair is to be reported. Although this could be calculated afterwards by comparing each clone pair to its representation in the previous revision, it is assumed that time can be saved by concluding changes from the suffix tree modification.

To enable mapping clone pairs on the one and reporting changes on the other hand, the notation of a clone pair is to be extended. Each clone pair is assigned a unique *ID* which allows locating the same clone pair in different revisions. Furthermore, each clone pair carries a tag which indicates the changes that happened to the clone pair from the previous to the current revision.

This section is organized as follows. Section 4.1. describes different changes that can happen to a clone pair between two revisions. Section 4.2 discusses why these changes are ambiguous. Section 4.3 outlines how tracing clone pairs integrates into *IDA* and how changes to clone pairs are derived from the suffix tree. Finally, Section 4.4 explains how remaining pairs are matched.

[4]http://www.nongnu.org/cvs/

4.1 Changing Clone Pairs

When the source code of a program changes from one revision to the next due to files that have been changed, clone pairs within these files might also be affected. Basically, there are three types of changes that can happen to a clone pair apart from the fact, that a pair can remain unchanged between two revisions.

A clone pair can be *added* from one revision to the next. The clone pair that is found in the current revision did not exist in the previous revision. This might be due to new files that are added to the program or the modification of existing files changing the sequence of tokens.

Furthermore, a clone pair can be *deleted*. The pair existed in the last revision, but cannot be found in the current revision. The deletion of a pair can be caused by the deletion of a file from the program or a change of the token sequence.

Beyond addition and deletion, a clone pair can be *modified* from one revision to the next. The clone pair existed in the previous and still exists in the current revision. However, the pair has changed in some way. Three different types of modifications that can happen to a clone pair are distinguished. They do not exclude each other, hence they can appear together for a single clone pair. The three types are the following.

4.1.1 Location Modification

A location modification has happened to a clone pair if the start or end position of at least one fragment has changed. The location of a fragment is modified if the value for start or end does not equal the previous value. Note, that if the location in terms of tokens changed, this does not necessarily mean, that the location also changed in terms of lines and vice versa. *IDA* considers both measurements. If the fragment's position has changed in tokens and/or lines, this is seen as a modification of the fragment's location. An example for a location modification can be seen in Figure 16.

4.1.2 Type Modification

The next type of modification refers to the type of the clone pair. When the values of tokens are changed within a fragment, this can lead to a change of the clone pairs type from type 1 to type 2 or the other way round. A

```
1 // Comment
2 do_it();
3 a = b + c;
4 name = "Peter";
```

(a) Old version

```
1 do_it();
2 a = b + c;
3 name = "Peter";
4 // Comment
```

(b) New version

Figure 16 – A fragment whose location was modified in terms of lines. Assuming, that the fragment consists of line 2 to 4 in the old version, the fragment spans lines 1 to 3 in the new version. Note that the location has not changed in terms of tokens because whitespace and comments are not considered to be tokens.

type modification is shown in Figure 17. One of the clone pairs' fragments is changed by renaming identifiers. This changes the clone pairs' type from 1 to 2.

```
do_it();
a = b + c;
a = 3 * a;
```

(a) $fragment_A$

```
do_it();
a = b + c;
a = 3 * a;
```

(b) Old version of $fragment_B$

```
do_it();
a = a + c;
a = 3 * b;
```

(c) New version of $fragment_B$

Figure 17 – A clone pair consisting of fragments A and B. When $fragment_B$ changes to the new version, the type of the clone pair is modified from 1 to 2.

4.1.3 Structural Modification

The remaining type of modifications is the one that changes the actual sequence of tokens of a fragment by inserting or deleting tokens. If a clone pair's structure is modified, the same tokens are inserted or removed from both fragments belonging to the clone pair. Regarding other works, this situation is referred to as a *consistent* change of the clone pair [ACP07, KSNM05, Kri07].

Summarizing the changes, the answer to Question 4 is, that a clone pair can be added, deleted, modified or remain unchanged from one revision to the next. Modification is further classified into type, location and structural modification. A simple taxonomy for changes of a clone pair is shown in Figure 18.

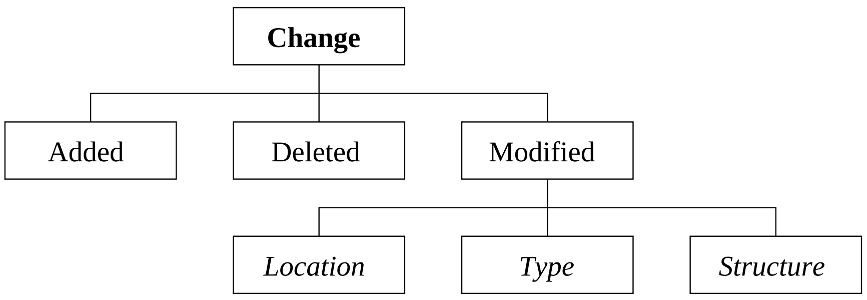


Figure 18 – A simple taxonomy for the changes that can happen to a clone pair between two revisions. Italic names indicate modifications that do not exclude each other.

4.2 Ambiguity of Changes

Unfortunately, changes of clone pairs are ambiguous, because the information, that the new version of a file is different from the old version does not yield the exact changes which have been made to the source code. By just analyzing two snapshots of a file, the actions that have really been performed by the author who changed the files are still unknown. The author might have deleted and rewritten everything between the two snapshots available, and from an external view, no change has occurred to the file. Though being quite obvious and not much of a problem in this simple case, more complex scenarios suffer from this lack of information.

Figure 19 shows two snapshots of code where the actual change cannot be reconstructed. Disregarding whitespace, the author could have inserted `call_bar();` at the end of the fragment. On the other hand, the last call `call_foo();` could have been deleted and `call_foo();` and `call_bar();` been inserted at the beginning. If lines 1 and 2 in the first revision form the fragment of a clone pair, it cannot be said whether this fragment is spanning lines 1 and 2 or lines 3 and 4 in the second revision.

```
1  call_foo();
2  call_bar();
3
4  call_foo();
```

(a) Revision 1

```
1  call_foo();
2  call_bar();
3  call_foo();
4  call_bar();
```

(b) Revision 2

Figure 19 – Two revisions of source code. What exactly happened from the first to the second revision can not be reconstructed.

Considering these ambiguities, any conclusions about clone pair changes are guesses and reflect the action which has most likely been performed by the author. The less changes to the source code an action implies, the more likely has the action been done by the author. Information about how this is quantified is given in Section 4.4.1. Based on the restricted information about changes to the source code, the actual action can neither be derived from suffix tree modifications nor from matching clone pairs afterwards.

4.3 Integration into *IDA*

This part explains how tracing clone pairs is integrated into *IDA*. Furthermore, it answers Question 5 by describing how clone pair changes can be derived during the incremental clone detection. The algorithm presented in Section 3 is slightly modified to allow tracing clone pairs. Whenever a clone pair was said to be deleted in Section 3, it rather has its status tag set to deleted but is not yet removed from the set of clone pairs. This is needed, because it might later turn out that the clone pair was not deleted but modified instead. The remaining part of this section describes how each of *IDA's* phases is extended.

After the analysis of the first revision and the creation of the initial set of clone pairs, each pair is assigned a unique ID. This allows to distinguish clone pairs based on their ID. Each clone pairs' status is set to *added*, because understandably, it has been detected for the first time in the first revision.

As for the preparations that have to be done for every revision, the set of clone pairs resulting from the last revision has to be prepared for the analysis of the current revision. This requires looking at each pair in the set. Every clone pair that has its status set to *deleted* is now finally removed from the list, as it has been reported as deleted in the last revision and is no longer needed. Every other clone pairs status is set to *unchanged* which reflects the default assumption, that nothing has happened at all.

When a deleted file is processed, all clone pairs from which at least one fragment is contained in the file, are tagged as *deleted*. They are not discarded yet, because they need to be reported as deleted in the post processing step for this revision. They are finally disposed in the preparation of the next revision as mentioned above. Addition of a file does not require looking at existing clone pairs, because until rerunning the *Baker* algorithm, no clone pair can exist of which a fragment is contained in the new file.

In the description of processing modified files, the concept of reusing edges has been introduced. Reusing an edge is the opportunity to derive changes to clone pairs of which one fragment relates to the edge's end node. Whenever an edge is reused, it is quite probable that the fragments which relate to the edge's end node are unchanged. As no modification has been done to the suffix tree, the sequence of cloned tokens (the path of the edge's start node) is unchanged and the distance from the end of the fragment to the end of the file remains the same.

However, the fragment might have moved, because tokens have been inserted before the fragment or tokens are located in different lines now. That is why for every fragment related to a reused edge's end node, the location is checked again in terms of tokens and lines. If values differ, a location modification is reported for the clone pair. Independent of whether the location changed, the fragment is made to reference the new version of the file, because the old version will be deleted later. After inserting the new version of the file, edges that are not reused and relate to suffixes of the old version, are removed from the suffix tree. For any edge deleted this way, the clone pairs related to this edge are marked as deleted.

Unfortunately, any other change to clone pairs cannot easily be concluded from the suffix tree modification. Minimal changes of fragments can move the respective nodes to a completely different location in the suffix tree. Assuming for example, that the first token of the fragments of a pair change, the new external nodes will be found in a completely different branch of the suffix tree. Any approaches to detecting these changes from the tree are far too time consuming, because in the worst case scenario the whole tree would have to be searched for matching nodes.

Figure 20 shows the generalized suffix tree with $\Delta = \{cabc\$_1, bcab\$_2\}$, which was obtained by inserting token c in both files and updating the suffix tree for $\Delta = \{abc\$_1, bab\$_2\}$ (see Figure 9b). Note how the nodes relevant for the clone pair ab (respectively cab after the modification) have moved in the tree.

During the post processing phase, the type of every clone pair which is not tagged as deleted, is checked. This is done by determining the current type of the pair and comparing it to its old type. If they are found to be different, a type modification is reported for that pair.

The remaining problem is, that a lot of clone pairs have been marked as deleted in the previous steps and many new clone pairs have been detected while rerunning the *Baker* algorithm, due to minimal changes in the sequence of tokens or location of fragments. Although it is technically correct to report these pairs as deleted and the new version as added, this does not reflect the real change that has happened. Instead, it is desirable to

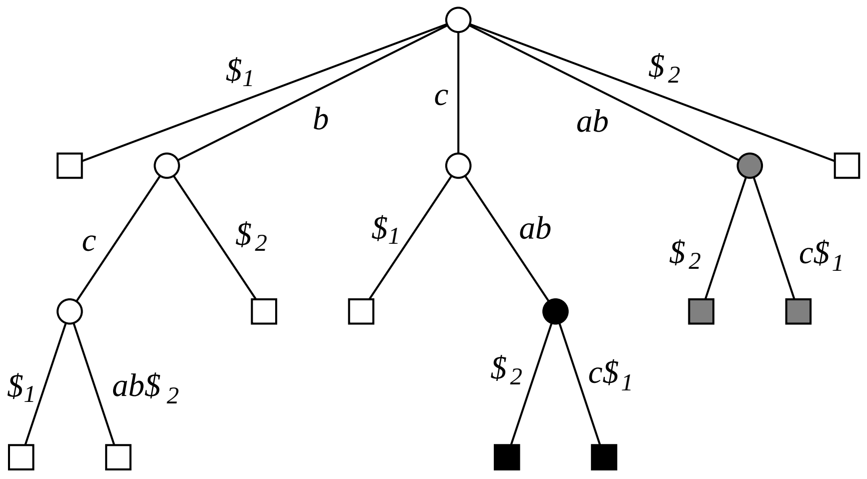


Figure 20 – Generalized suffix tree with $\Delta = \{cabc\$_1, bcab\$_2\}$. The grey colored nodes represent the clone pair with the token sequence *ab* as it was in the tree with $\Delta = \{abc\$_1, bab\$_2\}$. The structurally modified clone pair *cab* is represented by the black colored nodes. The nodes appear in a completely different branch in the tree, after the token *c* has been inserted in both files and the suffix tree has been updated.

have these clone pairs reported as modified instead of the old version being deleted and the new version added. The following section describes how *IDA* matches pairs after changed files have been processed to find likely modifications.

4.4 Matching Clone Pairs

Matching different source code fragments is not special to *IDA* and different approaches exist [KN06]. This section answers Question 6 by describing how *IDA* matches clone pairs of two consecutive revisions to find changes. The starting point are two sets of clone pairs. One contains all clone pairs from revision i which have been tagged as deleted. The other contains all clone pairs that have been newly detected in revision $i + 1$. The problem which is to be solved, is to find two clone pairs (one from each set) that are sufficiently similar according to some definition. These are reported as a modification instead of an addition and a deletion.

4.4.1 Distance Between Clone Pairs

The problem suggests a function to measure how similar two clone pairs are and answer the question how likely one pair is the modification of the other. From now on, this value is referred to as the *distance* between two clone pairs. The greater the distance, the less likely is the one pair a modification of the other. With CP being the set of all clone pairs, the required function is of type:

$$distance : CP \times CP \rightarrow \mathbb{N}_0$$

Due to the quadratic nature of the matching routine, there is a significant number of calls to this function for large clone pair sets. This imposes a strong restriction on the complexity of the function.

Two factors can be identified that contribute to the similarity of clone pairs. The first factor is the structure of the token sequence represented by the pairs. If the structure is almost the same, the assumption that one pair is the modification of the other, is probable. If on the other hand, the token sequences of both pairs show huge differences, the pairs are less likely a modification of each other. The second factor is the difference in location of the clone pair's fragments. The greater the difference between the locations of the fragments, the greater the distance between the two pairs. The value of the distance between two clone pairs is to be obtained by considering the difference in location and structure of the clone pair's fragments.

There are a number of methods that can be used for quantifying the difference between two strings of tokens. An easily applicable function is the *Hamming distance* [Ham50], counting the number of positions where the two strings differ. The problem of the Hamming distance is, that it cannot recognize shifted characters or tokens in a string. The distance between the strings `abcdef` and `fabcde` has the maximal value 6, although both strings share a common sequence of 5 characters. This makes the Hamming distance unusable as it does not represent the distance between clone pairs as required.

Another measure would be calculating the length of the longest common subsequence between the two strings. Like the Hamming distance, the result can easily be influenced by the insertion of single tokens. Given the strings `abcdef` and `abcgef`, the length of the longest common subsequence is 3, although the strings are identical except for one token.

A more sophisticated way to express the distance between two strings is the edit distance, counting the number of basic operations which must be performed to turn one string into the other. The values produced by the method presented by *Levenshtein* [Lev66] would much better resemble the structural differences between the token sequences. Unfortunately, the quadratic time complexity of the function calculating the Levenshtein distance makes the method unsuitable for the integration into *IDA*.

The only structural information about the token sequences, that requires constant time to be calculated, is the length. Therefore, the difference in length of the token sequences contributes to the difference of the clone pairs. Although this serves as a prototypical solution, future research is to be directed at finding a better solution to measuring the structural difference between two strings of tokens.

On the other hand, the difference in the fragments' locations can be retrieved in constant time. *IDA* adds up the difference between the starting positions of $fragment_A$ and the difference between the starting positions of $fragment_B$ for both pairs. The more the starting positions of the fragments differ, the greater is the distance between the two pairs. The difference of the end positions is not considered, because it is indirectly included by considering the length of the pairs. All differences are equally weighted, because it is assumed that the probability that a fragment moves equals the probability that the clone pair's length is changed. It is ensured, that the distance between a clone pair and itself is always 0. The function used by *IDA* is as follows.

$$\begin{aligned} distance(cp_1, cp_2) &= |start_{fragment_{A_{cp_1}}} - start_{fragment_{A_{cp_2}}}| \\ &+ |start_{fragment_{B_{cp_1}}} - start_{fragment_{B_{cp_2}}}| \\ &+ |length_{cp_1} - length_{cp_2}| \end{aligned}$$

It must be mentioned, that this function is by far not optimal. It might reveal changes although a fragment has been removed and a structurally completely different fragment just happened to be moved in its place. This indicates the need for more detailed information about changes. Information about changes based on files proves to be too imprecise.

4.4.2 Matching Procedure

For finding best matches between clone pairs, one has to keep in mind, that a match has to be best in both directions. Assume the following possible matches among clone pairs:

$$\begin{aligned}
distance(clone\_pair_0, clone\_pair_2) &= 12\\
distance(clone\_pair_0, clone\_pair_3) &= 37\\
distance(clone\_pair_1, clone\_pair_2) &= 2\\
distance(clone\_pair_1, clone\_pair_3) &= 142
\end{aligned}$$

If possible matches were processed sequentially, $clone\_pair_0$ would be reported as a modification of $clone\_pair_2$, because $clone\_pair_2$ is the best match for $clone\_pair_0$. The match between $clone\_pair_1$ and $clone\_pair_2$, which has a much lower distance, cannot be reported anymore, because $clone\_pair_2$ has already been matched with another clone pair. To avoid such situations, which can lead to obscure modifications of clone pairs, naive sequential processing cannot be done.

The method to find changed clone pairs processes as follows. First, all potentially deleted clone pairs are sorted into different sets. Each set contains all clone pairs for a specific combination of the files in which the fragments appear. A pair with the first fragment in file `sample.c` and the second fragment in file `test.c` is put into the set which collects all clones pairs between the files `sample.c` and `test.c`. It is worth recalling, that clone pairs are normalized and therefore the combination of the first fragment being in file `test.c` and the second in `sample.c` is impossible.

Like the deleted clone pairs, the new clone pairs are sorted into sets in the same way. This presorting is done to reduce the number of possible matches between clone pairs. The restriction is, that a change to a clone pair can only happen if the fragments stay in the same files. A modification of a pair where at least one fragment has moved to another file is always reported as a deletion of the old and an addition of the new pair.

After sorting, the sets of new clone pairs are sequentially processed. The pairs of each set are matched with the pairs of the corresponding set containing the deleted pairs. Matching is done by comparing each new pair to every old pair and calculating the distance between them. To significantly reduce the time needed by this step, a threshold has been introduced which represents the maximal distance two clone pairs are allowed to have. If the distance between any two pairs is above the threshold, the pairs are not considered as a possible match. The threshold is user-selectable, because no general statement can be made about how extensive changes of clone pairs can be for a specific program. A senseful value has to be chosen according to the nature of the program. For the tests presented in Section 5, a value of 100 has been chosen. Assume the example matches as given above with $clone\_pair_0$ and $clone\_pair_1$ being potentially deleted and $clone\_pair_2$ and $clone\_pair_3$ being potentially new.

Posterior to calculating the distances, all possible matches are sorted according to their distance. The list of possible matches is now as shown below. One possible match has been dropped, because the distance was above the threshold. The other matches have been sorted according to their distance.

$$\begin{aligned} distance(clone\_pair_1, clone\_pair_2) &= 2 \\ distance(clone\_pair_0, clone\_pair_2) &= 12 \\ distance(clone\_pair_0, clone\_pair_3) &= 37 \end{aligned}$$

Starting with the match that has the smallest distance, modifications are reported. The new clone pair adopts the ID of the old pair and has its status tag set to modified instead of added. Modifications of the pair are reported according to the comparison of the old pair to the new pair. If the location of the new pair is different from the old, a location modification is reported. If the type differs, a type modification has happened to the pair. Finally, the sequence of tokens is checked for equality and if not equal, a structural modification is reported for the pair. Concerning the example, $clone\_pair_2$ is reported as a modification of $clone\_pair_1$ and $clone\_pair_3$ is reported as a modification of $clone\_pair_0$.

A single clone pair can be contained in more than one possible match, but still can be matched only once. To prevent pairs from being matched more than once, every deleted clone pair that has been matched with a new one is marked an invalid. Whenever a pair marked as invalid is found, the possible match is not considered. In the example, the match between $clone\_pair_0$ and $clone\_pair_2$ is not considered, because $clone\_pair_2$ has already been matched before. When all clone pair sets have been processed, the overall set of clone pairs is updated with the new modifications.

4.5 *IDA* in Pseudocode

The different phases run by *IDA* have been explained in the previous sections. To serve as an overview over the concepts introduced in Sections 3 and 4, a pseudocode version of *IDA* is given. The pseudocode includes the actions required to allow tracing clone pairs. This is however only a rough outline to show the general structure.

```
Analyze revision 0 in the conventional way;

for revisions 1 .. n loop

   Discard all deleted clone pairs;
   Tag remaining clone pairs as unchanged;

   for every changed file loop

      if file is added then
         Create token table from new file;
         Insert file into suffix tree;

      elsif file is deleted then
         Tag all clone pairs related to file as deleted;
         Delete file from suffix tree;
         Discard files token table;

      elsif file is modified then
         Create token table from new version;
         Insert new version into suffix tree
           (if possible, reuse edges and derive changes);
         Delete old version from suffix tree;
         Discard token table of old version;
      end if;

   end loop;

   Check left extensibility of existing clone pairs;
   Find type modification of existing clone pairs;
   Run Baker to find new clone pairs;
   Filter new clone pairs;

   // Match clone pairs to find changes
   Partition deleted and new clone pair into buckets
   for each bucket
      Calculate distance for each combination of pairs;
      Sort possible matches according to their distance;
      Report changes starting with smallest distance;
   end loop;

   Integrate new clone pairs;
   Output list of clone pairs;

end loop;
```

Figure 21 – The ***I**ncremental **D**etection **A**lgorithm IDA* for analyzing $n+1$ revisions of a program.

5 Evaluation

In addition to specifying the algorithm, *IDA* has been implemented in the tool called *iClones (**i**ncremental **clones**)*. *iClones* is a derivation from the tool *clones* from the project *Bauhaus*. The infrastructure of *clones* was inherited and modified in the appropriate places. Apart from integrating the multi-revision analysis, the important data structures were modified to be useful for *iClones*. The main points were replacing the single token table with multiple token tables and extending the suffix tree to a generalized suffix tree.

This section describes the tests, that have been run with *iClones* in order to find and answer to Question 7 and show that the tool works as expected. Section 5.1 describes the test candidates on which *iClones* was run. Section 5.2 summarizes the tests that were done to check that *iClones* works as expected. Section 5.3 answers Question 7 by comparing *iClones'* to *clones'* performance.

5.1 Test Candidates

For testing the *IDA* implementation, three candidate programs have been chosen, which differ significantly in size. The criteria for choosing these programs were:

- The program's source code is easy accessible via the world wide web.
- Multiple revisions of the program are available (not only the latest release).
- The program has a significantly different size than the other test candidates.

The following sections describe the three candidates which have been chosen to compare *clones* to *iClones*.

5.1.1 *GNU Wget*

GNU Wget[5], from now on referred to as *wget*, is a small program for downloading files from the world wide web. The program is the smallest among

[5]http://www.gnu.org/software/wget/

Index	Revision	Date
0	*wget-1.11*	2008-01-26
1	*wget-1.11.1*	2008-03-24
2	*wget-1.11.2*	2008-04-30
3	*wget-1.11.3*	2008-05-29
4	*wget-1.11.4*	2008-06-29

Table 3 – *wget* Snapshots.

the three test candidates with approximately *25,000* SLOC[6]. The revisions used for testing are shown in Table 3.

5.1.2 *GNU Compiler Collection*

The middle-sized test candidate is the *GNU Compiler Collection*[7] *(gcc)*. In addition to the well-known compiler, the collection serves front-ends for several languages. For testing, only the core part of the collection is used (*gcc-core*), which consists of around *1,000,000* SLOC. The *gcc* source code is available via repository access or by downloading weekly snapshots. The revisions used for testing are shown in Table 4.

5.1.3 *Linux Kernel*

The third and largest among the candidates is the *Linux kernel*[8] *(kernel)*. The Linux kernel is a freely available operating system kernel, enhanced by huge amounts of hardware drivers. The kernel size ranges from *3,600,000* to *5,600,000* SLOC. The revisions used for testing are shown in Table 5.

5.1.4 Candidate Size

Table 6 compares the size of the three test candidates measured in SLOC. For each program, the minimum and the maximum number of SLOC is given. The difference in size among the test candidates can clearly be observed.

[6]Throughout this thesis, the size of programs is measured in *Source Lines of Code (SLOC)* using the program *SLOCCount* (http://www.dwheeler.com/sloccount), written by *David A. Wheeler.*

[7]http://gcc.gnu.org/

[8]http://www.kernel.org

Index	Revision	Date
0	*gcc-4.4-20080222*	2008-02-22
1	*gcc-4.4-20080229*	2008-02-29
2	*gcc-4.4-20080307*	2008-03-07
3	*gcc-4.4-20080314*	2008-03-14
4	*gcc-4.4-20080321*	2008-03-21
5	*gcc-4.4-20080328*	2008-03-28
6	*gcc-4.4-20080404*	2008-04-04
7	*gcc-4.4-20080411*	2008-04-11
8	*gcc-4.4-20080418*	2008-04-18
9	*gcc-4.4-20080425*	2008-04-25
10	*gcc-4.4-20080502*	2008-05-02
11	*gcc-4.4-20080509*	2008-05-09
12	*gcc-4.4-20080516*	2008-05-16
13	*gcc-4.4-20080523*	2008-05-23
14	*gcc-4.4-20080530*	2008-05-30
15	*gcc-4.4-20080606*	2008-06-06
16	*gcc-4.4-20080613*	2008-06-13
17	*gcc-4.4-20080620*	2008-06-20

Table 4 – *gcc* Snapshots.

Looking at Table 7, which shows the candidates' size in number of source files, one can observe, that the relation among the test candidates remains the same. For any of the following observations, only source files of the candidate programs are considered. Any files belonging for example to the build system or which are documentation are completely ignored.

5.1.5 Changes per Revision

Reading the descriptions about the three test candidates, one can observe that there is a significant difference in how and which revisions of each program are used. Needless to say, that *a revision* is everything but a precise measurement of changes to a program's source code. The amount of changes that happen between two revisions influences the time needed by *iClones*. Therefore, another look must be taken at the three test candidates. Figure 22 shows the percentage of source files that changed for each revision of the three programs. Revisions are referred to by an index starting from 0. The first revision is not shown, due to the fact, that there exists no previous revision from which files could have changed.

Index	Revision	Date
0	*linux-2.6.0*	2003-12-18
1	*linux-2.6.1*	2004-01-09
2	*linux-2.6.2*	2004-02-04
3	*linux-2.6.3*	2004-02-18
4	*linux-2.6.4*	2004-03-11
5	*linux-2.6.5*	2004-04-04
6	*linux-2.6.6*	2004-05-10
7	*linux-2.6.7*	2004-06-16
8	*linux-2.6.8*	2004-08-14
9	*linux-2.6.9*	2004-10-18
10	*linux-2.6.10*	2004-12-24
11	*linux-2.6.11*	2005-03-02
12	*linux-2.6.12*	2005-06-17
13	*linux-2.6.13*	2005-08-29
14	*linux-2.6.14*	2005-10-28
15	*linux-2.6.15*	2006-01-03
16	*linux-2.6.16*	2006-03-20
17	*linux-2.6.17*	2006-06-18
18	*linux-2.6.18*	2006-09-20
19	*linux-2.6.19*	2006-11-29
20	*linux-2.6.20*	2007-02-04
21	*linux-2.6.21*	2007-04-26
22	*linux-2.6.22*	2007-07-08
23	*linux-2.6.23*	2007-10-09
24	*linux-2.6.24*	2008-01-24
25	*linux-2.6.25*	2008-04-17

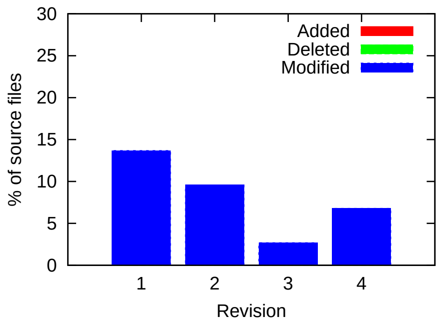
Table 5 – *kernel* Snapshots.

Candidate	Minimum	Maximum
wget	26	27
gcc	1,010	1,023
kernel	3,626	5,679

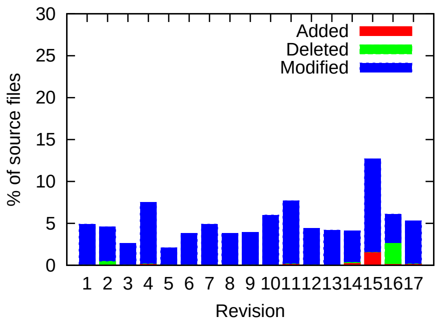
Table 6 – Size of the test candidates given in KSLOC.

Candidate	Minimum	Maximum
wget	73	73
gcc	1,929	1,978
kernel	12,432	19,464

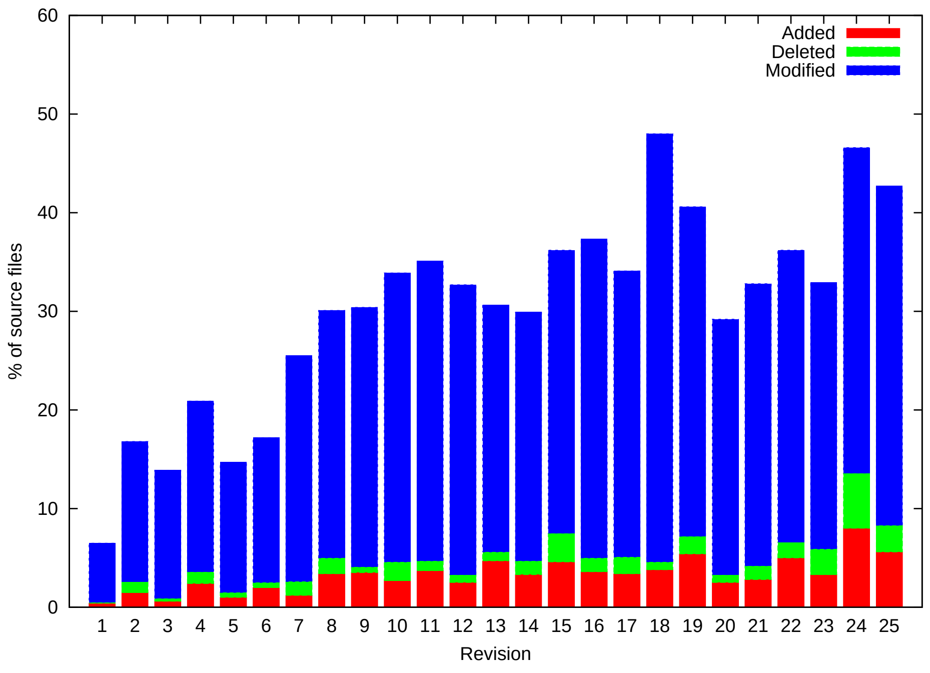
Table 7 – Size of the test candidates given in number of source files.



(a) *wget*



(b) *gcc*



(c) *kernel*

Figure 22 – Percent of source files that changed per revision.

It can be seen, that for *wget* and *gcc* 5% to 10% of the files change per revision, although more than a month is between *wget's* and just one week between *gcc's* revisions. This indicates, that the time between revisions is not useful for drawing conclusions about how many files change on average. In contrast to the other two candidates, the *kernel* has on average around 30% of its files changed per revision. But the time between revisions is much longer as with the other programs. Several months pass between the revisions of the kernel. Note, that these values are given at file granularity. These numbers are to be kept in mind when drawing conclusions from the tests which are described in the following.

5.2 Testing Correctness

Before *iClones'* performance can be evaluated, its results must be tested for correctness. Two things must be given for saying that *iClones* runs as expected and produces correct results.

- The set of clone pairs, which is reported after each revisions analysis, must exactly match the set of pairs revealed by *clones* (Section 5.2.1).
- The mapping of clone pairs between revisions must be correct in the way that for each clone pair in each revision, the status tag is set correctly (Section 5.2.2).

5.2.1 Correct Set of Clone Pairs

To show that *iClones* detects the correct set of clone pairs for each revision, the pairs have to be compared to a set of reference pairs. These reference pairs are obtained by running *clones* on each revision separately. When the respective pairs are compared to each other, no tolerance is permitted, because pairs are detected based on the same definition. The test was run in the following way: The first step consisted of running *clones* on every single revision of the three test candidate programs. The resulting clone pair sets were saved in order to be compared later. Then, *iClones* was run on the revisions of each candidate and the resulting set of clone pairs was also saved for each revision separately. Figure 23 shows how the set of clone pairs were determined and which of them were compared against each other.

Having available the two clone pair sets for every revision, one obtained by the conventional approach and the other by *iClones*, these were compared

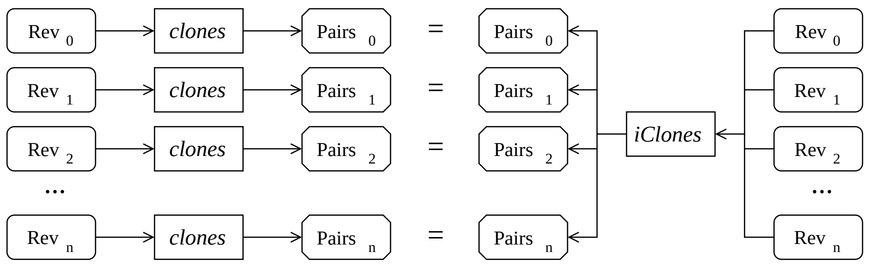


Figure 23 – Structure of how clone pair sets were compared against each other. Revisions $0 \ldots n$ have once been analyzed individually by the conventional approach and then by *iClones*. The resulting clone pair sets of each revision were checked for equality.

against each other to ensure, that they contain exactly the same clone pairs. As the sets of pairs are comparatively huge, the comparison was done via a script, that reads in the set of clone pairs, sorts them and does the actual comparison. Still, the minimum length of clones had to be chosen differently for the three candidates, to limit the size of the sets and keep them manageable for the script. The minimum clone length was set to 20 tokens for *wget*, 100 tokens for *gcc* and 300 tokens for the *kernel*.

Table 8 shows the sizes of the clone pairs sets that have been checked for equality for each revision. It was not only checked whether the sets are of the same size, but also if they contain exactly the same clone pairs. There is only one value given for the conventional approach and *iClones*, because both reported the same number of clone pairs for every single revision. Note, that these numbers just indicate how many clone pairs have been checked for equality. Any further interpretation of these numbers among the three test candidates is not possible, because different parameters to *iClones* were used for each test candidate.

The comparison has been done as shown in Figure 23 and *iClones* reports the same set of clone pairs for every revision as *clones*.

5.2.2 Correct Mapping

Knowing that *iClones* produces the same set of clone pairs is on its own not enough to say that *iClones* works correct. What remains to be checked is that the mapping of clone pairs from one revision to the next is correct. The mapping is correct, if the status tag for each clone pair resembles the actions which have been done by the author between two revisions. However, considering the ambiguity of changes described in Section 4.2, it is impossible to test whether the mapping is correct or not. Instead, it is checked whether

Revision	0	1	2	3	4	5	6	7	8
wget	4246	4253	4256	4254	4256	–	–	–	–
gcc	297082	297089	297090	298011	298014	298089	298438	298430	298489
kernel	13362	13341	13881	13485	14013	14238	14040	14328	15022

Revision	9	10	11	12	13	14	15	16	17
gcc	298490	300468	300463	300586	300601	300601	299321	299297	299610
kernel	15477	14749	14309	14451	17728	16710	17361	18421	18538

Revision	18	19	20	21	22	23	24	25
kernel	18769	18017	19558	22002	20881	22220	25178	38142

Table 8 – The number of clone pairs found in every revision of the candidate programs. *clones* and *iClones* report the same number of pairs for each revision.

the mapping is senseful and reflects the most likely actions performed by the author.

Analogous to testing the correctness of clone pair sets, one could think of comparing the mapping to a reference mapping. However, *Kim et al.* already stated in their discussion about matching program elements, that

> *"It is difficult to evaluate matching techniques because there is no reference data set or archive of editing logs."* [KN06].

Two kinds of tests have been done in order to get an indication that the mapping produced by *iClones* is senseful. During development of *iClones*, around 50 small artificial test cases have been created to test whether the program works as expected. Because these test cases were crafted by hand, the expected result could be manually determined and formulated. The results produced by *iClones* were compared to the expected outcome in an automated way. 15 of these test cases have been used for regression testing throughout the development phase.

In addition to artificial test cases, some clone pairs from a real world program were tested. *iClones* was run on the three test candidate programs and random samples have been selected from the resulting clone pair sets. These samples have been manually checked for soundness. The reported changes of the pair were verified and adjacent clone pairs were checked in order to be sure, that the reported changes are senseful. The complexity of this process limited the number of samples that could be checked manually. For every candidate, five random samples have been chosen, that reported different combinations of changes. All samples were found to report a correct and senseful change of the respective clone pairs.

There is no doubt, that these test do not suffice to show that the mapping produced by *iClones* is correct or at least every reported change is senseful. For further research and drawing conclusions from the mapping, large-scale automated tests are required. This could either be done by somehow formulating when a change is assumed to be senseful, or by comparing the resulting mapping to some reference mapping. *Kim et al.*, for example, published results produced by the *Clone Genealogy Extractor* [KSNM05]. Problems are that these data are given in a completely different format and further investigation is needed to say whether the mappings are comparable at all. The limited tests that have been performed do however indicate that the mapping produced by *iClones* is correct and senseful.

5.3 Testing Performance

The main reason for developing *IDA* and implementing it in *iClones* is to reduce the time consumption for clone detection in multiple revisions of a program. This section answers Question 7 by comparing the performance of *iClones* to *clones*. This is not straightforward, as there are a number of parameters which influence the performance. The most significant impact on the performance is the number of clone pairs which are found and need to be traced. To outline this influence, every test is run twice. Once with a relatively small and once with a relatively large number of clone pairs compared to the program's size. The amount of clone pairs found is regulated by adjusting the minimum length for clone pairs. The same parameters were used for *clones* and *iClones* to make the results comparable.

The following sections present the test results for each of the three test candidates. Figures show the time consumption split into the different phases which are the following:

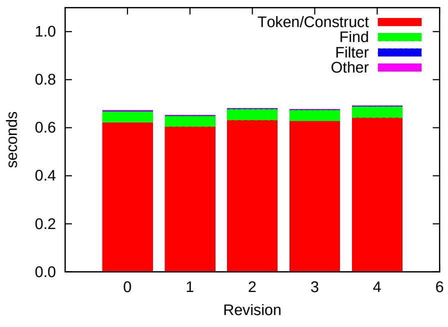
- **Token/Construct**: This phase includes tokenizing files which are added as well as new versions of files which are modified. In addition, any modification to the suffix tree resulting from the changed files contributes to this phase.
- **Find**: Within this phase, *Baker's* algorithm is run to extract maximal matches from the modified suffix tree.
- **Filter**: Existing and potentially new clone pairs are filtered and those that do not match the filtering criteria are discarded.

- **Match**: Potentially deleted and potentially added clone pairs are matched in order to find changed clone pairs. This phase does not appear in the conventional *clones* approach and is therefore not shown in the respective figures.
- **Other**: This phase includes anything not included in any of the other phases.

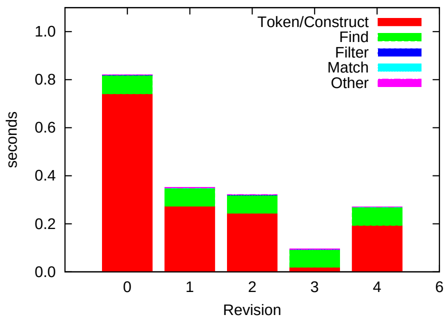
5.3.1 *wget*

This section shows how *IDA* performs when run on the revisions of *wget*, managing just a few (Figure 24) and a large number of clone pairs (Figure 25). It can be seen, that the conventional approach needs around $700ms$ for each revision with no significant difference between the revisions. However, there is a noticeable difference in the time needed by *iClones* for each revision. Revision 0 takes more time to analyze in comparison to *clones*. On the other hand, every following revision is analyzed much faster. They only require 15% to 50% of the time needed by *clones*. Adding up the first two revisions for both approaches, the sum is already smaller for *iClones*. This means as soon as more than one revision is analyzed, *iClones* needs less time than *clones*.

Another aspect one can observe is, that the time needed by *iClones* to analyze a single revision is proportional to the percentage of files that changed from the last revision. Comparing Figures 24b and 25b to Figure 22a, it can be seen that the pattern among revisions 1 to 4 is the same.



(a) *clones*

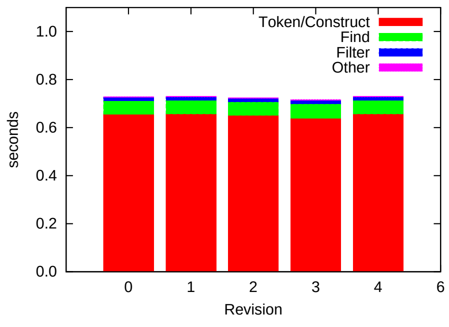


(b) *iClones*

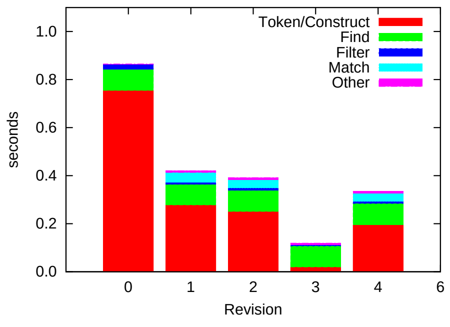
Figure 24 – Performance comparison of *clones* and *iClones* run on the revisions of *wget*. The minimum clone length was set to 40 tokens ($\approx$ 4 LOC) resulting in around 600 clone pairs per revision.

When the number of detected clone pairs is increased, the picture stays almost the same. Each phase needs a slightly increased amount of time due to the larger number of clone pairs that are processed. The relations among the revisions do not change.

One more thought should be given to revision 3 of *wget*. In this revision, only two files have been modified. Observing Figures 24b and 25b, the time required to analyze this revision is almost completely induced by running *Baker's* algorithm to find new clone pairs. All other phases contribute only marginally to the time consumption. This situation suggests, that the integration of *IDA* into an IDE and running the algorithm for every single changed file is feasible.



(a) *clones*



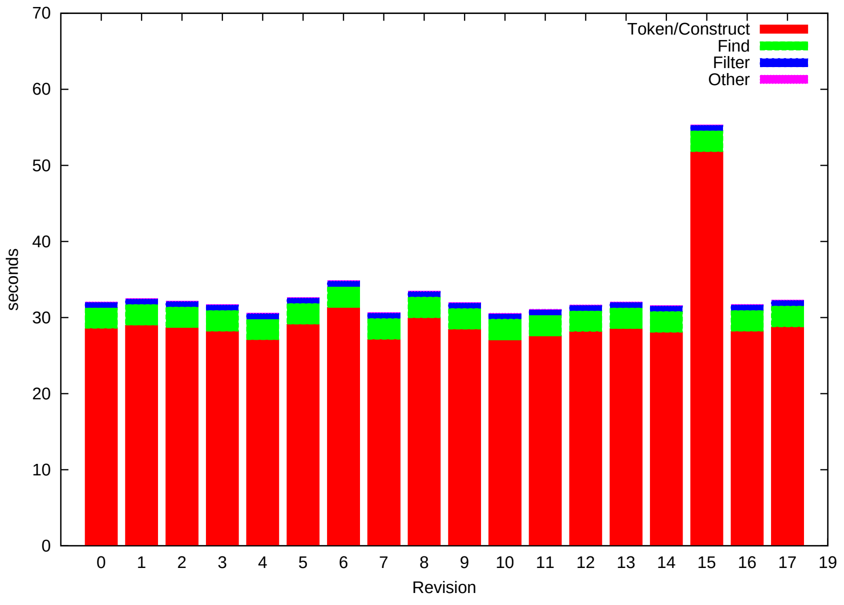
(b) *iClones*

Figure 25 – Performance comparison of *clones* and *iClones* run on the revisions of *wget*. The minimum clone length was set to 15 tokens (≈ 1.5 LOC) resulting in around 15,000 clone pairs per revision.

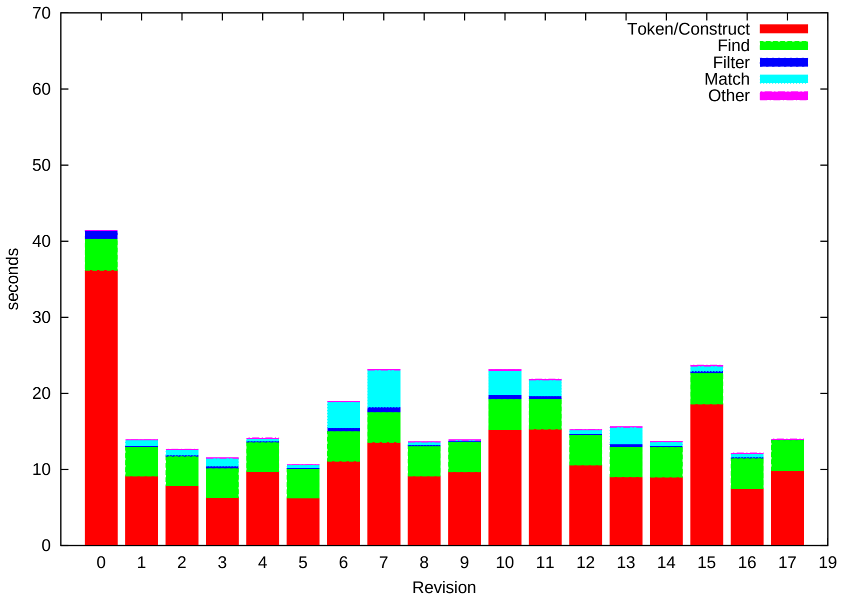
5.3.2 *gcc*

The second test candidate is *gcc* which is significantly larger than *wget*. Looking at Figure 26, the same pattern as with *wget* can be observed. The increased time usage for revision 15 results from a bug in the version of *clones* that was used for testing.

Analogous to *wget*, the analysis of the first revision takes more time with *iClones* than with *clones*. However, any other revision is processed much faster. In comparison to *wget*, the filtering phase is more obvious, because the size of the clone pair set is much larger. Apart from some exceptions, the time needed for filtering, is proportional to the overall time of the respective revision. When comparing the overall time of each revision to the percentage of changed files from Figure 22b, similarities in the pattern can be detected.



(a) *clones*



(b) *iClones*

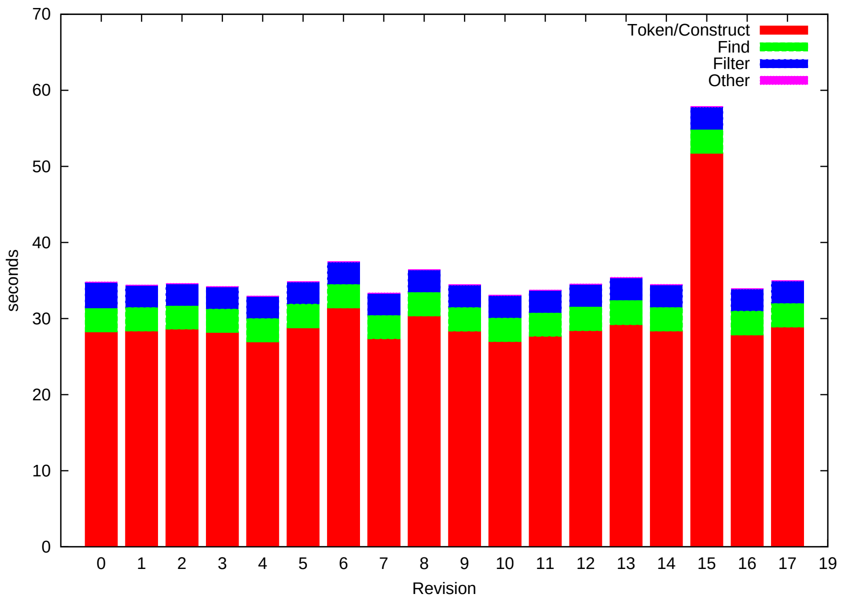
Figure 26 – Performance comparison of *clones* and *iClones* run on the revisions of *gcc*. The minimum clone length was set to 200 tokens (≈ 20 LOC) resulting in between 10,000 and 30,000 clone pairs per revision.

Increasing the size of the clone pair set by reducing the minimum length of clone pairs results in a different picture. Observing Figure 27 it can be seen, that the time needed for each phase except the matching phase increases slightly. On the other hand, the time usage of the matching phases rises dramatically. This is due to the fact, that many more clones pairs need to be matched against each other. However, the increase is not dependent on the percentage of files changed, but rather on the number of clone pairs, that are affected by the changed files. This can be seen by comparing revisions 7 and 15. Figure 22b shows that in revision 15 more than twice as many files changed than in revision 7, but still the matching phase of revision 7 takes almost three times as long as the one of revision 15.

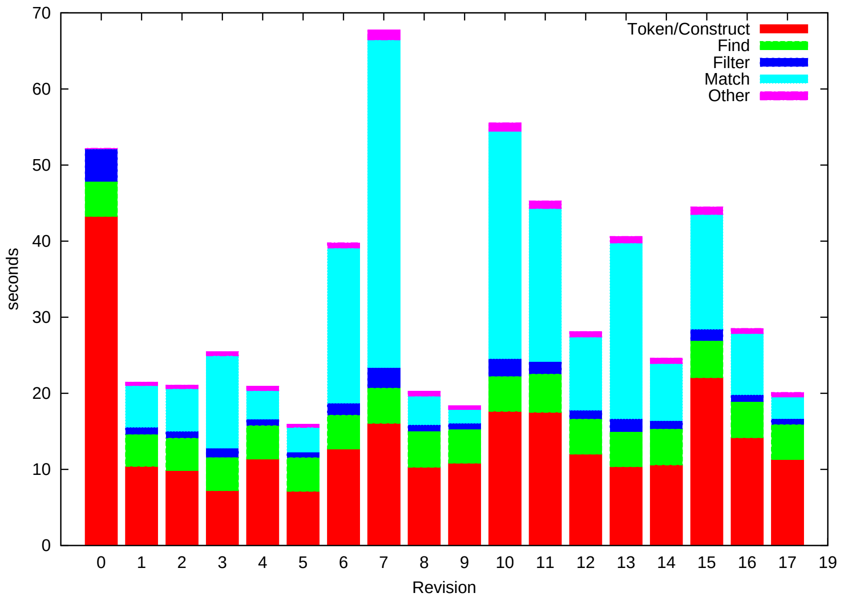
5.3.3 *kernel*

The third and last candidate on which *iClones* was tested is the Linux *kernel*. Apart from being the largest program in terms of SLOC, the *kernel* is different from the other two scenarios, because of having a large percentage of its files changed per revision. Figure 28 shows how this influences the time consumption. In contrast to the other candidate programs, the first revision requires noticeably more time than with *clones*. Although the next few revisions need less time with *iClones* in comparison to *clones*, every other revision requires considerably more time. Instead of an extended matching phase, the excessive time consumption results from an elongated tokenizing and construction phase. There are several reasons for this:

- In contrast to both other candidates, the *kernel* has a huge amount of files changed for the later revisions. In addition, a noticeable amount of files is added for each revision. Every new and modified file needs to be tokenized, requiring a great share of the overall time. Considering that around 35% of the source files are added or modified for each revision, 35% of the *kernel's* source code needs to be tokenized for each revision.
- The number of added, deleted and modified files is also responsible for the time required to update the suffix tree. The more changed files need to be processed, the more costly is the suffix tree update. The huge amount of changed files requires extensive modification of the suffix tree.
- The linear growth of the *kernel's* size is not to be neglected. Observing Figure 22c, one can observe, that in every revision more files are added than deleted. Starting from around 3,600,000 SLOC, the *kernel's* size increases to 5,600,000 SLOC. Needless to say, that the time needed



(a) *clones*



(b) *iClones*

Figure 27 – Performance comparison of *clones* and *iClones* run on the revisions of *gcc*. The minimum clone length was set to 40 tokens ($\approx$ 4 LOC) resulting in between 1,000,000 and 1,300,000 clone pairs per revision.

to analyze a revision depends on the revision's size. This can also be observed in conjunction with *clones* observing Figures 28a and 29a.

All three factors contribute to the increased time usage of the tokenizing and construction phase. In contrast to the previous test candidates, the extended matching phase does not carry as much weight as shown in Figure 29.

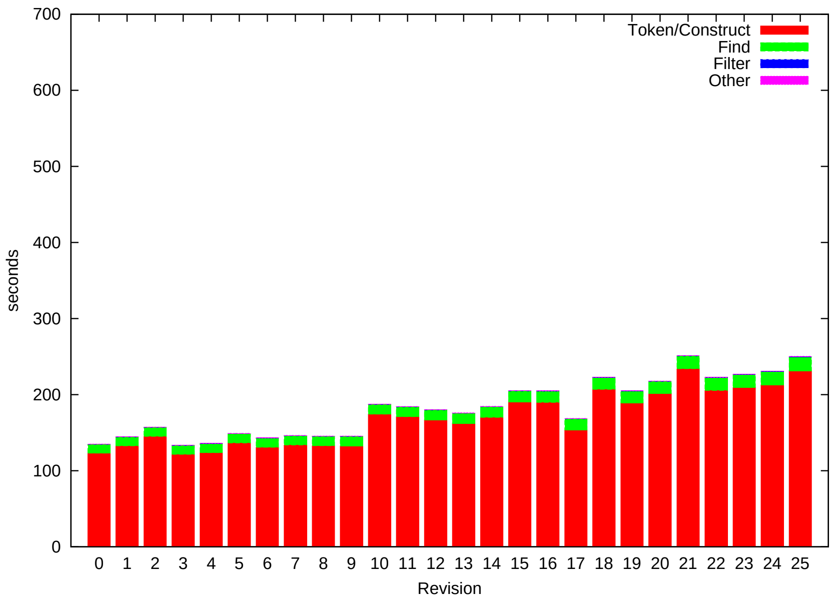
5.4 Summary

All three test candidates show, that independent of the number of detected clone pairs, *iClones* needs more time than *clones* to analyze the first revision of a program. This overhead is caused by using multiple token tables and a generalized suffix tree. Using multiple token tables requires two indirections instead of a single one. Although this is just a minimal increase for a single access, the intense usage of the token tables makes the overhead noticeable.

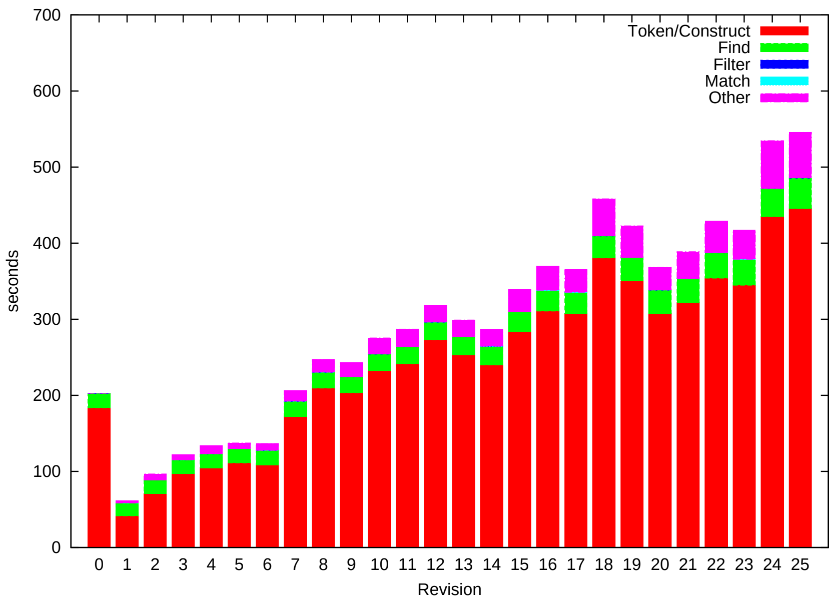
Furthermore, the increased complexity of the suffix tree contributes to the increased time consumption. Maintaining the links for keeping edge labels consistent is done, although only a single revision is to be analyzed. Whether this overhead is acceptable for analyzing a single revision of a program depends on the specific application.

In addition to causing a calculation overhead, the extended data structures cause *iClones* to require more memory than *clones*. This is due to internal references, which on the one hand are required to assure consistency, and on the other hand allow faster processing. The bi-directional links for keeping edge labels consistent as described in Section 3.2.2 are an example for additional references. The reference from an external node to the clone pairs which relate to this node is another one. Still, the memory consumption is proportional to the size of the revision which is analyzed. As everything related to a specific revision is discarded upon analyzing the next revision, no artifacts remain which could possibly lead to an increased memory consumption with every new revision, independent of the revision's size.

Regarding the tests which have been presented in this thesis, *iClones* needs around three times as much memory compared to *clones*. When analyzing revision 1 of *gcc*, *clones* needs around $550MByte$, whereas *iClones* requires $1500MByte$. For analyzing revision 1 of the *kernel*, both tools require four times as much memory compared to analyzing *gcc*. There are two reason for the increased memory consumption of *iClones*. As mentioned above, data structures have been extended to be reused. Additional references and links require more space. The other reason is, that *iClones* cannot discard any

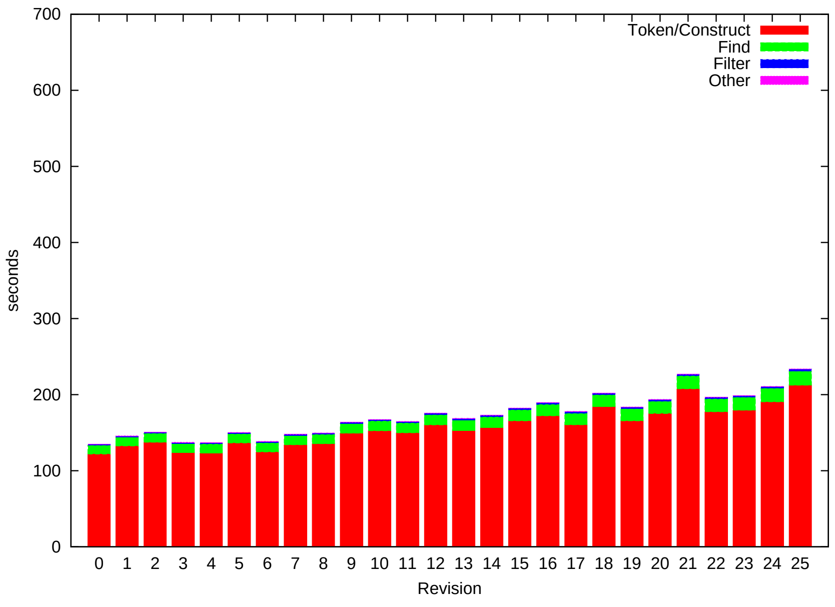


(a) *clones*

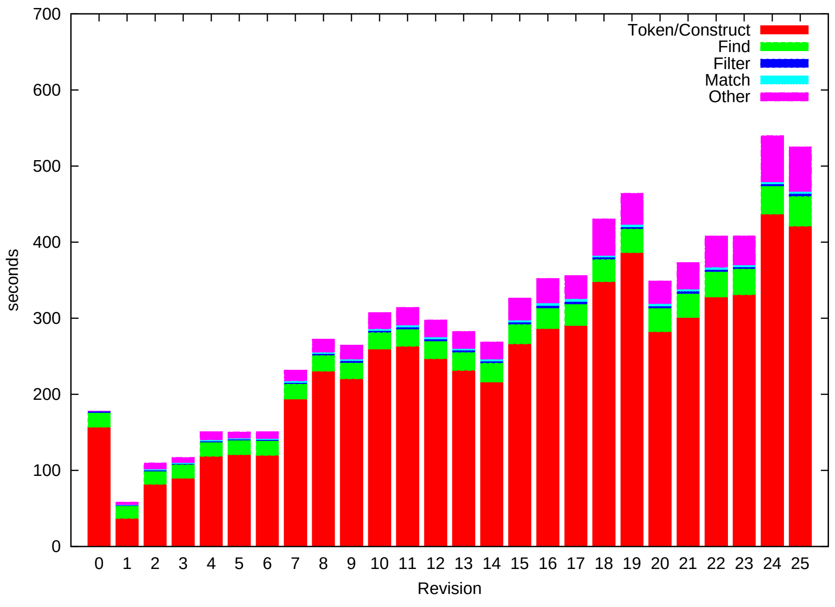


(b) *iClones*

Figure 28 – Performance comparison of *clones* and *iClones* run on the revisions of the *kernel*. The minimum clone length was set to 300 tokens ($\approx$ 30 LOC) resulting in between 10,000 and 40,000 clone pairs per revision.



(a) *clones*



(b) *iClones*

Figure 29 – Performance comparison of *clones* and *iClones* run on the revisions of the *kernel*. The minimum clone length was set to 200 tokens (≈ 20 LOC) resulting in between 200,000 to 800,000 clone pairs per revision.

data structures, because they are to be reused for the analysis of the next revision. *clones* can discard any intermediate data structures once they are not needed any more. It does not hold all data structures in memory at the same time. *iClones* on the other hand is not able to free memory by releasing intermediate data structures. This leads to a noticeably increased overall memory usage.

6 Conclusion

This section interprets the evaluation results presented in the last section and describes some thoughts for future development. Any conclusions and numbers are based on the tests that have been run and presented in this thesis. Further evaluation is needed to verify that these numbers and conclusions are generally valid. Section 6.1 draws conclusions from the test results for the different phases of *iClones*. Part 6.2 summarizes *IDA* and Section 6.3 presents perspectives for future work. Finally, Section 6.4 names some possible applications for *IDA* and *iClones*.

6.1 *iClones* and its Different Phases

Section 5.3 showed, that no general statement can be made about how much time *iClones* can save in comparison to *clones*. Different factors affect the time consumption of different phases and contribute to the overall time usage. The following sections answer Question 8 by describing which factors affect *iClones'* phases.

6.1.1 Tokenizing/Constructing

The time needed for tokenizing depends on the amount and size of files that changed in the respective revision. The more SLOC the changed files contain, the more time is needed to parse the files and build the token tables for them. This is nothing special to *iClones* and the time contribution is relatively low. However, *iClones* has an obvious advantage towards *clones*, because it just needs to tokenize new and modified files, whereas *clones* needs to tokenize all files of every revision.

The modification of the suffix tree on the other hand, needs a little more investigation. The test results showed, that this phase depends on the number of changes made for the current revision. There is a definite time advantage of *iClones* towards *clones* if the percentage of changed files stays within a limited range. Testing with *wget* and *gcc* showed, that with around 10% of changed files, modifying the suffix tree is noticeably faster than rebuilding the complete tree. On the other hand, when around 20% and more of the files change, there is no obvious benefit in modifying the tree. These numbers do however heavily depend on the system on which the program is run and the number of SLOC contained in the files.

6.1.2 Finding Maximal Matches

The time needed to run *Baker's* algorithm depends only on the size of the suffix tree and the number of clone pairs found. During the analysis of multiple revisions and given that the program does not dramatically change, the size of the suffix tree stays almost the same, although files are added, deleted and modified. This results in an almost constant time usage by this phase. *iClones* needs 30% to 40% more time than *clones* due to the additional indirection caused by the use of multiple token tables and the test whether edges are added or not.

6.1.3 Filtering

The time required for filtering depends on the size of the clone pair set and the number of potentially new clone pairs extracted by the previous phase. Like with tokenizing, the share of the overall time which relates to the filtering phase is fairly small. In most cases, *iClones* needs to filter less pairs than *clones*, because the set of new clone pairs which are extracted from the suffix tree is smaller than the set of all clone pairs which needs to be filtered by *clones*.

6.1.4 Matching

This phase has the most influence on the overall time consumption of the tool *iClones*. Similar to the construction phase, one might assume the time usage is proportional to the percentage of files changed. However, the time in fact depends on the number of clone pairs which are affected by the changed files. When many clone pairs for which at least one fragment stems from a modified file are marked as deleted, the number of possible matches for potentially new clone pairs increases. Due to the quadratic complexity of the matching algorithm, every new clone pair needs to be compared to more deleted clone pairs and the time usage rapidly increases.

6.1.5 Other

This phase depends on the number of changes that are made. As it contains mostly preparations and clean-up actions, more changes lead to a higher time consumption. This phase becomes noticeable with an increasing number of changes from one revision to the next, but still has limited influence on the overall time.

6.2 Summary of Results

Within this thesis, the incremental clone detection approach *IDA* has been presented. In contrast to conventional clone detection approaches, *IDA* analyzes multiple revisions of a program. The benefit compared to separate clone detection for each revision is, that intermediate data structures can be reused for the analysis of the next revision. This avoids analyzing parts of the source code again and again which do not change between revisions. Using information about the files that changed from the previous to the current revision, *IDA* modifies the token tables, the generalized suffix tree and the set of clone pairs to conform to the current revision. The result produced by *IDA* is a set of clone pairs for each revision that is analyzed.

In addition to only creating independent sets of clone pairs, a mapping is created between the clone pairs of two consecutive revisions. To make clone pairs traceable across revisions, each clone pair is assigned a unique ID. If the clone pair is found in another revision, it is either unchanged, or some kind of modifications has happened to the clone pair. Possible modifications are a change in location, a changed type or a structural change. Wherever possible, changes to clone pairs are derived from the modifications that are done to the suffix tree.

Remaining clone pairs, whose changes could not be derived from the suffix tree, are subsequently checked for modifications. New clone pairs are matched with possibly deleted clone pairs in order to find modified clone pairs. This has to be done in a post processing step, because already slight modifications in a clone pairs token sequence inhibit concluding the change from the suffix tree. After matching has been done, the changes that happened to each clone pair are reported.

IDA has been implemented in the tool *iClones*. *iClones* is a derivation of the tool *clones* from the *Bauhaus* project. *iClones* has been evaluated by comparing its performance to *clones*.

6.2.1 *iClones* and *clones*

iClones needs considerably less time than the existing token-based approach *clones* for analyzing multiple revisions of a program when less than 10% of the program's files changed between two revisions. Results show that given a limited amount of files that change per revision, *iClones* is already faster in analyzing only two revisions. With any following revision, this advantage becomes more and more explicit. How much time is saved compared to *clones* depends mainly on the number of files that change per revision.

What could also be seen is, that *iClones* requires noticeably more time to analyze the first revision of the program. This indicates, that *clones* has an advantage when only a single revision of a program is to be analyzed.

Unfortunately, no large-scale tests could be run to verify, that the mapping between clone pairs produced by *iClones* is correct and senseful. Due to the absence of benchmarks [KN06], only limited test cases could be used to obtain an indication, that *iClones'* results are correct and useful. Further investigation is needed to solve this issue.

6.3 Future Research

During this thesis, some problems related to *IDA* and *iClones* have been mentioned which could not be solved within this thesis. This section briefly describes ideas that might help to improve *IDA* and *iClones* but could not yet be considered. These will serve as a basis for future work on *IDA* and its implementation.

6.3.1 Testing the Mapping

The mapping between clone pairs created by *iClones* could not be tested on a large scale for correctness. To make the mapping usable and base further assumptions on it, the soundness of the mapping has to be verified. This could either be done by checking the mapping according to an editing log or comparing it to reference data obtained by some other method.

6.3.2 Rerunning *Baker's* algorithm

Though running *Baker's* algorithm is comparatively fast, there is still room for improvement. If it was known, that large sub-trees of the suffix tree do not contain any new edges and due to the length criteria cannot form new clone pairs, these sub-trees could be left out by the algorithm. However, one has to make sure that the overhead for managing the information which sub-trees contain new edges, does not exceed the time which can be saved by leaving out portions of the tree.

6.3.3 Parameterized Detection

In its current version, *IDA* supports the detection of unparameterized clone pairs. *IDA* might be extended to allow parameterized clone detection, revealing only type-2 clone pairs with consistent changes. This requires combining the concepts of parameterized strings and generalized suffix trees. The impact of removing or adding a parameterized string to Δ on the parameterized generalized suffix tree must be investigated. If the impact stays in a limited range, parameterized clone detection can be integrated into *IDA*.

6.3.4 Saving Intermediate Results

iClones still needs a considerable amount of time to produce its results, although large improvement has been made compared to *clones*. As new revisions become available, it might be rather painful to start analyzing everything from the very beginning. One could think of offering a facility, that can store the internal state of *iClones* and its data structures on disk. Whenever new revisions become available, *iClones* could reload this state and proceed from where it stopped last time.

6.3.5 Develop a Customized Output Format

Current output formats implemented in *clones* and *iClones* are not designed for saving information about clone pairs across multiple revisions. This leads to a lot of data being represented redundantly. One could think of a new format for saving information about clone pairs, which tries to eliminate this redundancy and leads to a significantly smaller file size.

6.3.6 Finding Split Clones

A change that can likely happen to a clone pair is, that one of its fragments is split because tokens are inserted. *IDA* reports only situations as a structural change where the same tokens have been inserted into both fragments. If however different tokens are inserted, or the insertion takes place at different locations, this is not recognized by *IDA*. Effort could be spent on finding and reporting these situations.

6.3.7 Sophisticated Distance Function

The distance function currently used by *IDA* to determine how likely a clone pair is the modification of another pair is not optimal. Further effort is to be spent on finding a function whose output resembles more the authors actions. The time constraint is however to be kept in mind. The function should require constant or at most linear time.

6.3.8 Using Parallel Processes

To speed up the matching phase, *IDA* could be extended in order to use parallel processes. As every clone pair is just contained in a single set and the sets are processed independently, the work could be delegated to more than one processor.

6.3.9 Detailed Information About Changes

Probably the most promising improvement would be considering changes based on lines or even tokens instead of files. This would most likely not accelerate the modification of the suffix tree, but allow for much more precise information about modifications of fragments. Knowing that the lines of a fragment did not change, no structural or type modification can wrongly be reported for the respective clone pair.

Additionally, time can be saved by the extra information, because clone pairs whose fragments are not changed, do not need to be checked for structural and type modification. The calculation of the location modification would also be straightforward.

One could also think of reusing token tables of changed files. Based on the information which lines of the file changed, only parts of the token table would need to be updated.

6.4 Applications for *IDA*

There are two applications of *IDA* envisioned. One is integrating *IDA* into an IDE to do "on-line" clone detection. The other is to aid in investigating the evolution of clones.

Integrating *IDA* into an IDE allows the user to get up-to-date information about clones. *IDA* runs as a background process and holds the token tables, the suffix tree and the set of clone pairs for the program that the user edits. When the user modifies a file of the program, the new state of the program is assumed to be a new revision of the program with a single file changed. This new revision will be processed as described in this thesis. *IDA* does not require a lot of time processing the revision, considering that only a single file has changed. *IDA's* results inform the user about how and which of the clone pairs were affected by his actions.

The other possible application of *IDA* is to assist in investigating the evolution of clones. Clones can be detected for multiple revisions of a program requiring less time than separate clone detection for each revision. By using the mapping that *IDA* produces, the lifetime of each clone pair can be retrieved. Starting from simple questions like

- What is the average lifetime of a clone pair?
- How many clone pairs are added and how many deleted per revision?
- How often is the structure of a clone pair changed?

diverse quantitative and qualitative analyses can be run using *IDA's* results. After all, understanding the evolution of clones will contribute to answering the question if clones are to be considered harmful or not.

A Supplying Data to *iClones*

For searching clone pairs in different revisions of a program, *iClones* needs access to the source code of every revision which is to be analyzed. *iClones* expects a single directory which contains one subdirectory for each revision of the program. Because directories are processed in alphanumerical order, the naming scheme should reflect the order in which the revisions are to be analyzed.

In addition to the source code for each revision, *iClones* needs a list of changes which happened from the last revision to the current one. These changes need to be summarized in a file named `changes`, which must be located in the top-level directory of each revision. Note, that the first revision does not need such a file, because every source code file is assumed to be added. Figure 30 shows the scheme for the directory structure. The directory `base` is passed as an argument to *iClones*.

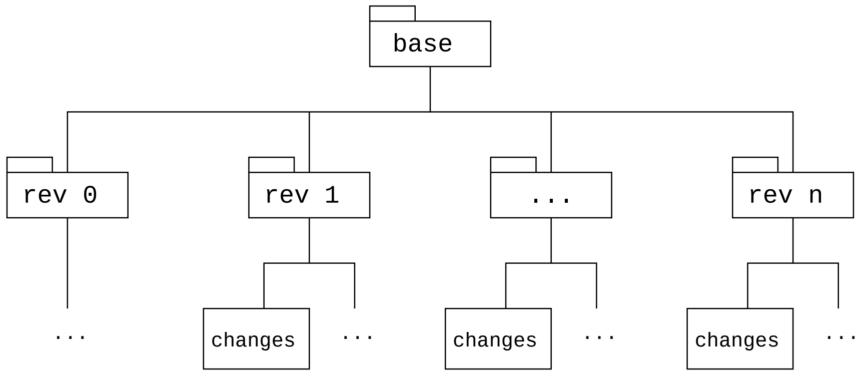


Figure 30 – Scheme for the directory structure which is expected by *iClones*.

As stated above, changes are supplied in a file called `changes` for every revision. The format of this file follows the format used by *subversion*[9], for example when running the command `svn status`. Each line represents a change to one single file. A line consists of a character identifying the type of the change and the path to the file which has been changed relative to the revisions top-level directory, separated by a space character. The following changes are supported.

- `A`: Indicates, that the respective file has been added in this revision.
- `D`: The corresponding file has been deleted in this revision.

[9]http://subversion.tigris.org/

- `M`: The file has been modified between the last and the current revision

Lines starting with the `#` character are ignored and can be used for comments. The order in which changed files are given is not important, as every change is processed on its own. A sample file is shown in Figure 31.

```
# Comments are ignored
M src/util/serializer.c
M src/main/main.c
A src/util/printer.h
A src/util/printer.c
```

Figure 31 – A sample file, containing information about changes.

Storing all files of every revision requires huge amount of disk space, especially for large programs. *iClones* actually only needs the files which have changed in the current revision. All other files are ignored and therefore do not need to exist. Note, that the changed files still need to be found under the same directory path. Collecting all changed files in a single directory is not supported by *iClones*.

B Extended Output Format

Several formats exists in which information about clone pairs and clone classes can be presented to a user. Unfortunately, all formats supported by *clones* are designed for a set of clones found in a single revision of the program. The need for a new format that can represent clones across revisions has been mentioned in Section 6.3.5. Currently, *iClones* extends one of the formats used by *clones*.

One of the formats to describe clone pairs is the *clone pair format* (*cpf*). The format carries the following information about each clone pair. The two fragments of the clone pair are each specified by the *file*, *start line* and *end line* in which the fragments appears. In addition, the *type* of the pair and its *length* in tokens as well as in lines is given. A small sample *cpf* is given in Figure 32.

```
test/single/1.c  1  2  test/single/2.c  4  5  1  2  2
test/single/2.c  3  4  test/single/3.c  1  2  1  2  2
test/single/1.c  1  2  test/single/2.c  2  3  2  2  2
test/single/2.c  2  3  test/single/2.c  4  5  1  2  2
```

Figure 32 – Sample output of clone pairs in the *cpf*. Each line contains the file, start and end line of the first fragment, file start and end line of the second fragment, the type and the length in lines and tokens of the clone pair.

Unfortunately the format is not distinct enough to embody all information produced by *iClones*. Two extensions have to be made to the format in order to be useful for *iClones*. First, each clone pair is given a unique ID. This is important in order to trace a clone pair across revisions.

The second extension is a status tag for each pair describing which changes happened to the clone pair from the last to the current revision. Together with its ID, the nature of the change can be obtained. Currently, the following status tags are used by *iClones*.

- `+`: The clone pair has been added in this revision. It was not present in the previous revision.
- `-`: The clone pair has been deleted. It was present in the last revision but does no longer exist.
- `T`: The location of at least one of the fragments from the clone pair has changed in terms of tokens.

- `L`: Same as `T` using lines instead of tokens.
- `Y`: Indicates, that the type of the clone pair has changed.
- `S`: A structural modification happened to the fragments of the clone pair.
- If no tag is given, the clone pair has not changed in any way from the last to the current revision.

Note, that `T`, `L`, `Y` and `S` can appear in any arbitrary combination.

A sample output of *iClones* in the extended *cpf* for a single revision is shown in Figure 33. Note, that the second pairs' fragments still relate to `rev0`, because the pair has been deleted and is not found in `rev1`. Every pair that is removed has to be reported as deleted in the successive revision, because during the analysis of the last revision in which the pair is contained, it cannot be known that the pair will be deleted.

```
1  test/multi/rev1/1.c  1  2  test/multi/rev1/2.c  4  5  1  2  2  TL
2  test/multi/rev0/2.c  3  4  test/multi/rev0/3.c  1  2  1  2  2  -
3  test/multi/rev1/1.c  1  2  test/multi/rev1/2.c  2  3  2  2  2
4  test/multi/rev1/2.c  2  3  test/multi/rev1/2.c  4  5  1  2  2  +
```

Figure 33 – Sample output of clone pairs in the extended *cpf*. Each line contains the ID, file, start and end line of the first fragment, file start and end line of the second fragment, the type, the length in lines and tokens and the modification tag of the clone pair.

Other output formats are currently not supported, but could be extended in a similar fashion.

List of Figures

List of Tables

References

[ACP07] L. Aversano, L. Cerulo, and M. Di Penta. How clones are maintained: An empirical study. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 81–90, Washington, DC, USA, 2007. IEEE Computer Society.

[ACPM01] G. Antoniol, G. Casazza, M. Di Penta, and E. Merlo. Modeling clones evolution through time series. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 273–280, Washington, DC, USA, 2001. IEEE Computer Society.

[AFG$^+$93] A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. Dynamic dictionary matching. *Journal of Computer and System Sciences*, 49(2):208–222, 1993.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Boston, MA, USA, 1986.

[AVMP02] G. Antoniol, U. Villano, E. Merlo, and M. Di Penta. Analyzing cloning evolution in the linux kernel. *Information and Software Technology*, 44(13), 2002.

[Bak93] B. S. Baker. On finding duplication in strings and software. Technical report, AT&T Bell Laboratories, 1993.

[Bak95] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Second Working Conference on Reverse Engineering*, pages 86–95, Washington, DC, USA, 1995. IEEE Computer Society.

[Bak97] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal on Computing*, 26(5):1343–1362, 1997.

[Bel07] S. Bellon. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[BMG06] M. Balint, R. Marinescu, and T. Girba. How developers copy. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 56–68, Washington, DC, USA, 2006. IEEE Computer Society.

[BYM$^+$98] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of* *the International Conference on Software Maintenance*, pages 368–377, Washington, DC, USA, 1998. IEEE Computer Society.

[CDS04] J. R. Cordy, T. R. Dean, and N. Synytskyy. Practical language-independent detection of near-miss clones. In *Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 1–12. IBM Press, 2004.

[DBF$^+$95] N. Davey, P. Barson, S. Field, R. Frank, and D. Tansley. The development of a software clone detector. *International Journal of Applied Software Technology*, 1(3–4):219–236, 1995.

[DER07] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *Proceedings of the 29th International Conference on Software Engineering*, pages 158–167, Washington, DC, USA, 2007. IEEE Computer Society.

[DRD99] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the International Conference on Software Maintenance*, pages 109–118, Washington, DC, USA, 1999. IEEE Computer Society.

[EFM07] W. S. Evans, C. W. Fraser, and F. Ma. Clone detection via structural abstraction. In *Proceedings of the 14th Working Conference on Reverse Engineering*, pages 150–159, Washington, DC, USA, 2007. IEEE Computer Society.

[FGM97] P. Ferragina, R. Grossi, and M. Montagero. A note on updating suffix tree labels. In *Proceedings of the Third Italian Conference on Algorithms and Complexity*, pages 181–192, London, UK, 1997. Springer-Verlag.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[GLS92] D. Gusfield, G. M. Landau, and B. Schieber. An efficient algorithm for the all pairs suffix-prefix problem. *Information Processing Letters*, 41(4):181–185, 1992.

[Ham50] R. W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 26(2):147–160, 1950.

[HK08] J. Harder and R. Koschke. Empirische Grundlagen für das Klonmanagement. In R. Gimnich, U. Kaiser, J. Quante, and

A. Winter, editors, *10th Workshop Software Reengineering*, volume 126 of *GI Lecture Notes in Informatics*, pages 127–133. GI, 2008.

[JMSG07] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[Joh94] J. H. Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the International Conference on Software Maintenance*, pages 120–126, Washington, DC, USA, 1994. IEEE Computer Society.

[KAG$^+$07] C. Kapser, P. Anderson, M. Godfrey, R. Koschke, M. Rieger, F. van Rysselberghe, and P. Weißgerber. Subjectivity in clone judgment: Can we ever agree? In R. Koschke, E. Merlo, and A. Walenstein, editors, *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings, Schloss Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik.

[KFF06] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the 13th Working Conference on Reverse Engineering*, pages 253–262, Washington, DC, USA, 2006. IEEE Computer Society.

[KG06] C. Kapser and M. W. Godfrey. "Cloning considered harmful" considered harmful. In *Proceedings of the 13th Working Conference on Reverse Engineering*, pages 19–28, Washington, DC, USA, 2006. IEEE Computer Society.

[KH01] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56, London, UK, 2001. Springer-Verlag.

[KKI02] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[KN05] M. Kim and D. Notkin. Using a clone genealogy extractor for understanding and supporting evolution of code clones. *SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.

[KN06] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In S. Diehl, H. Gall, and A. E. Hassan, editors, *Proceedings of the 2006 International Workshop on Mining Software Repositories*, pages 58–64, New York, NY, USA, 2006. ACM.

[Kon97] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Proceedings of the 4th Working Conference on Reverse Engineering*, pages 44–54, Washington, DC, USA, 1997. IEEE Computer Society.

[Kri01] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering)*, pages 301–309, Washington, DC, USA, 2001. IEEE Computer Society.

[Kri07] J. Krinke. A study of consistent and inconsistent changes to code clones. In *Proceedings of the 14th Working Conference on Reverse Engineering*, pages 170–178, Washington, DC, USA, 2007. IEEE Computer Society.

[KSNM05] M. Kim, V. Sazawal, D. Notkin, and G.l Murphy. An empirical study of code clone genealogies. *SIGSOFT Software Engineering Notes*, 30(5):187–196, 2005.

[Lev66] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. Technical Report 8, 1966.

[LLM06] Z. Li, S. Lu, and S. Myagmar. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.*, 32(3):176–192, 2006.

[LWN07] A. Lozano, M. Wermelinger, and B. Nuseibeh. Evaluating the harmfulness of cloning: A change based experiment. In *Proceedings of the 4th International Workshop on Mining Software Repositories*, pages 18–21, Washington, DC, USA, 2007. IEEE Computer Society.

[McC76] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[MLM96] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the 1996 International Conference on Software Maintenance*, pages 244–253, Washington, DC, USA, 1996. IEEE Computer Society.

[PMDL99] J.-F. Patenaude, E. Merlo, M. Dagenais, and B. Laguë. Extending software quality assessment techniques to java systems. In *Proceedings of the 7th International Workshop on Program Comprehension*, pages 49–56, Washington, DC, USA, 1999. IEEE Computer Society.

[Ukk95] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[Wal07] A. Walenstein. Code clones: Reconsidering terminology. In R. Koschke, E. Merlo, and A. Walenstein, editors, *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings, Schloss Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik.

[WM05] R. Wettel and R. Marinescu. Archeology of code duplication: Recovering duplication chains from small duplication fragments. In *Proceedings of the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 63–70, Washington, DC, USA, 2005. IEEE Computer Society.

[WSvGF04] V. Wahler, D. Seipel, J. Wolff v. Gudenberg, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *Proceedings of the 4th International Workshop on Source Code Analysis and Manipulation*, pages 128–135, Washington, DC, USA, 2004. IEEE Computer Society.

[Yan91] W. Yang. Identifying syntactic differences between two programs. *Software-Practice & Experience*, 21(7):739–755, 1991.