Clone Detection Using Dependence Analysis and Lexical Analysis

Final Report

Yue JIA 0636332

Supervised by Professor Mark Harman

Department of Computer Science

King's College London

September 2007

Acknowledgments

I would like to express my sincere gratitude to a number of people who ever offered their invaluable help in this project.

My deepest gratitude goes first and foremost to my supervisor, Professor Mark Harman, who generously gave me his insightful instructions and earnest help all through the design, implementation and evaluation stages, paved a smooth way for the proposal of this project to come into being.

Second, I want to thank Nicolas Gold, Dave Binkley, Tao Jiang and other members from CREST who were present in my presentations and gave their significant suggestions that helped me better my project.

Then I'd like to give my thanks to Stefan Bellon from Stuttgart University who provided the additional experimental data of other clone detection tools.

My appreciation also goes to Toshihiro Kamiya, the developer of CCFinder, from Osaka University, whose advice and useful information helped a great deal.

I am also greatly indebted to my friends, Daniel and Jing, who carefully proofread this paper and suggested many important modifications of grammar and syntax so that my work is expressed clearly.

Finally, special appreciation is extended to my parents and Grace for their unconditioned love, endless support and heartfelt blessing over these years.

Without the support and help of the above mentioned persons, I could not have accomplished this project, so let me give them my warmest thanks once again.

Abstract

A software clone is a code fragment identical or similar to another in the source file. Since clone code is considered as one of the main problems to reduce the maintainability of software, several clone detection techniques have been proposed, some of them are very fast but only good at detecting normal clones, and some others are slow but can find advantaged clones. However none of them can detect all kinds of clone efficiently.

This project proposes a new approach to detect software clones in large software systems as an aid to maintenance and re-engineering. The novel aspect of our approach is it takes the advantages of textual and semantic type detection techniques, which performs lexical analysis and dependence analysis together during the detection process. The key benefit of this approach is that it improves both of the precision and performance of clone detection at the same time.

This paper also presents an experiment that evaluates seven clone detectors including our prototype based on six large C and Java programs. We studied 325,935 submitted clone pairs in all, which are based on our clone coverage evaluation method, and focus on studying the difference between each tool. The experiment result shows that our technique can scale nearly as well as the fast techniques but can give precision and recall nearly as good as the precise techniques.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software Maintenance plays an important role in the life cycle of a software product. It involves changes to the software in order to fix bugs, add new features, adapt to new environment, and improve the structure of the program [Ede93]. However, maintenance is a difficult activity which is generally considered to be expensive and time-consuming. According to a study of 487 companies [LS80], it is estimated that over 70% of the total efforts in the software life cycle go to maintenance activities.

The difficulty of software maintenance is usually caused by the poor structure of the source code. The poor structure is sometimes due to poor initial design of the system or lack of necessary features in the development tools, but often because of poor programming activities during the implementation. The focus of our work is to design an efficient tool that can deal with one of the most common reasons of poor program structure, software clones.

1.1 Software clones

Clone code or code duplication is a code portion in software source files that is identical or similar to the other [KKI02]. It is a form of software reuse, and exists in almost every software project. The results of several studies [KH01b, Bak95, DRD99] indicate that a considerable fraction (5-10%) of the source code in a large software systems is duplicate code.

Software clone is usually generated by programmers' copy and paste activities. When system needs some new functionalities, programmers often copy and paste an existing similar working fine code with slight modifications. Clone code can also be introduced by re-implementing the existing module followed by previous source code. These duplication activities are usually not documented, although it seems to be a simple and effective method.

Software clone has a number of negative effects on the quality of the software. Besides increasing the amount of the code which needs to be maintained, duplication also increases the defect probability and resource requirements [Rie05]. The following list gives an overview of these problems:

- **Increasing maintaining works**: When programmers maintain a piece of clone code, the changes should also perform on every other clone pairs. Since programmers who usually have no records of these duplicate code, the maintaining work should perform on the entire system.
- **Increasing defect probability**: By simply copying a piece of code into a new context, which will cause the conflict between each other, e.g. conflict and clash between variables from the copied code and variables in the new context. Dependencies of copied code may also not be fully understood by the new context is another potential defect cause.
- **Increasing Resource Requirements**: The clone code will consume more compilation times, because more codes have to be compiled. It may also lead the upgrading of hardware resources, especially when the system is running in a tight hardware environment, e.g. telecommunication switch, which the software system upgrading will lead an upgrading in hardware as well.

Concerning the negative effects of clone code, numerous clone detection techniques have been proposed, which can be used as a solution to find the software clone in source code. Table 1.1 lists the mainstream clone techniques. According to their code representation, the detection technique can be roughly classified into five categories, which are text–based, token–based, abstract syntax tree (AST)–based, program dependence graph (PDG)–based, and metrics–based approach. Each of them has their own specials (e.g. text-based method is very fast but can not find the advanced clone, which can be detected by PDG–based method, one takes very long time), however all of them are limited to the trade off between the precision and performance.

Table 1.1: List of mainstream techniques for detecting clone code

Reference	Code Representation	Comparison Technique
[Joh94]	Substrings	String Matching
[Bak92]	Parameterized Strings	String Matching
[DRD99]	Token	Token Matching
[KKI02]	Token	Token Matching
[MM01]	Token	Latent Semantic Analysis
[BYM+98]	Abstract Syntax Tree	Tree Matching
[BMD+99]	Abstract Syntax Tree	Tree Matching
[Lei]	Abstract Syntax Tree	Hybrid, Syntax Driven
[Kri01]	Program Dependence Graph	Graph Matching
[KH01a]	Program Dependence Graph	Backward slicing

Table 1.2: Software clones detected by our prototype

Code fragment (a)

```
++  public ServiceType getExecutor(){
          JavadocType.Handle javadocType =
            (JavadocType.Handle)getProperty(PROP_ EXECUTOR);
++     JavadocType type = null;
++     if (javadocType != null){
++       type = (JavadocType)javadocType.getServiceType();}
++     if (type == null){
           if (isWriteExternal()){
             return null;}
++        return(JavadocType)Lookup.getDefault().lookup(org.netbeans.modules.
++          javadoc.settings.ExternalJavadocSettingsService.class);}
++     return type;}
```

Code fragment (b)

```
++  public ServiceType getExternalExecutorEngine(){
++     ExternalJavadocExecutor service = null;
++     if (executor != null){
++       service = (ExternalJavadocExecutor)executor.getServiceType();}
++     if (service == null){
++      return (ServiceType)Lookup.getDefault().lookup(org.netbeans.modules.
++        javadoc.ExternalJavadocExecutor.class);}
++     return service;}
```

Code fragment (c)

```
++  public ServiceType getSearchEngine(){
          JavadocSearchType.Handle searchType =
            (JavadocSearchType.Handle)getProperty(PROP_ SEARCH);
++     JavadocSearchType type = null;
++     if (searchType != null){
++       type = (JavadocSearchType)searchType.getServiceType();}
++     if (type == null){
           if (isWriteExternal()){
             return null;}
++        return (JavadocSearchType)Lookup.getDefault().lookup(org.netbeans.modules.
++          javadoc.search.Jdk12SearchType.class);}
++     return type;}
```

1.2 Project aim and object

This project focuses on detecting clone code in a large software system as an aid to maintenance and re-engineering. The aim is to design and initially implement a tool for detecting clones. The novel aspect of the work is using lexical and dependence analysis together on detection process, which can scale nearly as well as the fast techniques but also can give precision and recall nearly as good as the precise techniques. The following list gives the main objective.

- **Precision**: The software clone can be classified into different types according to the level of the modification. Some advanced clones (e.g. type 3) are very hard to be detected by general method (see section 2.1). Our approach should support to detect all kinds of clones.
- **Performance**: Our approach involves dependence analysis which is not only a highly precise technique but also an expensive way. To reduce the running time and memory consumption at the same time is another objective.
- **Scalability**: Scalability is a very important factor of the project. Our approach should be able to scale up to very large (at least 100 K LOC) real software system.

Table 1.2 shows a real example, which is detected by our clone detection prototype from a java project (Netbeans–Javadoc). The three code fragments are selected from three different files and the duplicated code are indicated by "++" signs. As the duplicated code are not exactly same,(i.e. some variable names and types are changed, some lines are deleted, and some new lines are also added), these types of clone are difficult to be detected by general approach, however it can be detected by our approach in an acceptable amount of time.

1.3 Outline

The rest of this report is organized as follows. Chapter 2 introduces the terminology and background for clone detection at first, and then describes the different existing clone detection techniques. Chapter 3 discusses the relevant issues of mainstream detection techniques and provides the good characters for a good clone detection approach. Chapter 4 defines the detail design of the main detection algorithm. Chapter 5 provides the detail design and implementation of the prototype. Chapter 6 describes the experiment method for our evaluation, then lists and analyzes the experiment results in detail. Chapter 7 provides directions for future work, as well as the conclusions of this report.

Chapter 2

Literature Review

2.1 Basic Concepts of Clone

Although copy and paste is recognized as a common reason for software clones, there is still no solid definition of what constitutes a clone [LLWY03]. The general answer of "what is clone?" is that two code fragments if they are identical or similar [KKI02]. However the different definitions of similarity levels lead to different degrees on clone definitions.

According to the different similarities, the concept of clone can be classified into two categories: One type definition of similarity considers the program text level, which if two code fragments form a clone pair, their source code texts must be same or similar. The other considers the semantic level, which the clone code must have the same behaviors, in other words, by giving the same initial conditions, in which they must have the similar post conditions.

As the software clones are most often the result of copy and paste, in this project, the term of "clone" is defined as describing an association between two fragments that are considered as copied, and the two fragments of code is called a clone pair.

Definition 1 *if code fragment f_1 or code fragment f_2 is copied from the other, the pair (f_1, f_2) forms a clone.*

Definition 2 *a code fragment is a triple (f, s, e) which consists of the source file f, the start line s and the end line e of a portion of the code.*

According to research [BMD +99], as detected clones are usually directly used by programmer or system maintainer, so the clone should be presented as a group, not a separated clone pair. The clone pair can be grouped together by their common property into a clone class (see Figure 2.1).

After copy and paste activities, the duplicated code can be changed according to programmer's need, that the types of changes may include insertions and deletions of lines, or modifications with the line. By different types of changes, the clones can be distinguished as following types:

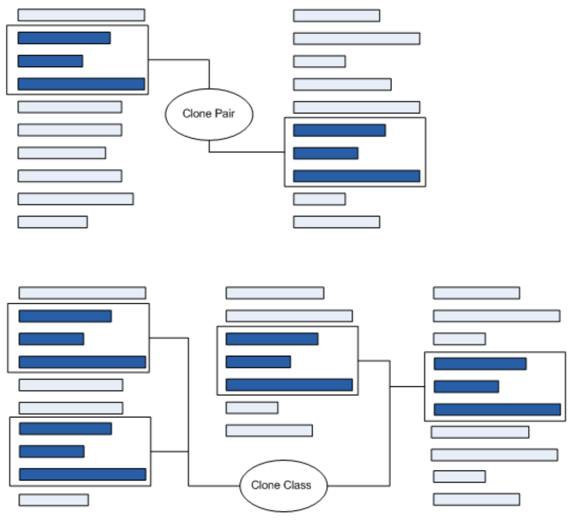


Figure 2.1: Clone pair and clone class

- **Type 1**: is exact copy without any modifications, i.e. two fragments of code are exactly same. This kind of clone can be detected easily.
- **Type 2**: is syntactically identical copy, which the modification is within a line of code, e.g. changes of the modifier and variable name, that will not affect the structure of the code fragment. With some basic transformation processes, this kind of clone can be also detected easily.
- **Type 3**: is copy with further modifications. E.g. a new statement can be added, or some statements can be removed. The structure of code fragment may be changed and they may even look or behave slight differently. This kind of clone is hard to be detected , because the fully context understanding is needed.

Another type of clone which is called parameterized clone is proposed by Baker[Bak92]. It is a subset of type 2 clones, that two code fragments f_1 and f_2 are parameterized clone pair if mapping from f_1's identifiers to f_2's identifiers and the mapping result is equal to fragment f_2.

As type 1 and type 2 clones are precisely defined, they can be easily detected. The definition of type 3 clones is still vague, although someone considers a gap between two any type of clone forms the type 3 clone, and others define if base on a threshold of the Levenshtein Distance, i.e. the value of modification steps for transforming one string into another. In this project, the type 3 clone is defined as that a limited gap value between any type of clone (include type 3) forms a type 3 clone.

2.2 Clone Detection Techniques

Clone Detection techniques can be used to fight the software clone in source file. Although there are varieties of clone detection techniques existing(see Table 1.1), the main processes of them are similar. The difference between them is their data representation, which the basic unit used to describe the source code. According to the data representation the mainstream clone detection techniques can be classified into five categories, as shown in Table 2.1.

Table 2.1: List of five categories of mainstream clone deletion techniques

Category	Code Representation	Comparison Technique
Text–based	Text	String Matching
Token–based	Token	Token Matching
Metrics–based	Text	Metrics vector
Abstract Syntax Tree(AST)–based	AST	Subtree Matching
Program Dependence Graph(PDG)–based	PDG	Subgraph Matching

Although some of these clone detection techniques' data representation and matching algorithm are different, they all follow a general process, which can be roughly broken down into five phases;

1. Code Partitioning: Preprocess the original source code from input. Filter the uninterested information at first, and then break down the code according to code representation's character.
2. Transformation: Transfer the source code into the proper code presentation according to specific rules.
3. Comparison: Use the characteristics of the code representation to carry on comparison of each other, then aggregate result as clone pair.
4. Filtering: Filter uninterested clone pair from the previous result following the filtering rules.
5. Aggregation: Group up similar clone pairs into clone class.

Among those processes, the comparison step is the most time consuming phase, that affects the performance of the detection technique mainly and the precision is depended on the information from the source code, which is defined by the code representation. Since the comparison algorithm is also depended on accessibility of the code representation, the code representation is the main component of a clone detection tool.

2.3 State of the Art

Software clone detection is an active field of research, since 1992 numbers of clone detection techniques have been proposed[Bak92]. The following section summarizes the characters of each type of mainstream detection techniques.

- **Text–based technique**

  Text–based technique is the oldest and simplest way to detect clone, which takes each line of source code as code representation[Bak92, Bak95]. In order to increase the performance, lines are often transformed by a hash function and uninterested code, such as comments and white spaces are filtered. The result of comparison is presented in a dot plot graph, where each dot indicates a pair of cloned lines. A clone pair can be determined as a sequence of uninterrupted diagonals line of spot.

  Because text–based technique does not perform any syntactical or semantically analysis on source code, it's one of the fastest clone detection approaches. It can easily deal with type 1 clone, and with additional data transformation, the type 2 can also be taken care. However without information of syntactical or semantically level support, the third type of clone can not be detected at all.

- **Token–based technique**

  Token–based technique is similar to text–based technique, however, instead of taking a line of code as representation directly, a lexical analyzer converts each line of code into a sequence of token[KKI02]. After data values and identifier are substituted by some special tokens, the token sequences of lines are compared efficiently through a suffix tree algorithm. The result is also presented in dot plot graph.

  This technique is slightly slower than text–based method, because of the tokenization step. However, applying suffix tree matching algorithm, the time complexity is similar as text–based technique. By breaking line into tokens, it can easily detect both type 1 and type 2 clone, with token filter applied, the result of clone can be controlled very precisely, for example, skip any uninterested information.

- **Metric–based technique**

  Metric–based technique gathers different metrics from a particular code fragments, such as, a function or a class, then groups these metric together into a metrics vector. After that it compares these metric vector instead of actual code directly[LPM$^+$97, KDM$^+$96], because this method is focused on a specific type of code fragments, it can only detect an type of high level clone, e.g. duplicated function.

- **Abstract Syntax Tree(AST)–based technique**

  AST–based technique uses a parser to obtain a syntactical representation of the source code, that typically an abstract syntax tree(AST), at first. Then it computes the hash code of each subtree, and compares result with others to find the similar subtrees in the AST. The clone pair can be extracted from the found similar subtrees[BYM$^+$98].

By using AST as code representation gives this technique an better understanding of the system structure, that can be used to obtain any type of clone and the hash value of each statement of each branch can be compressed together by a better hash algorithm, where will reduce the complex time from $O(n^3)$ to $O(n)$. However parsing source file is still a very expensive process on both time and memory .

- **Program Dependence Graph(PDG)–based technique**

  Control and data dependencies of a program can be represented as a program dependency graph. As it records the relationship between the data and structure, it can be used to tracing the modification after programmer's copy and paste activities[KH01b, KH01b, Kri01]. The PDG–based technique takes one step further than AST–based method, that to obtain the PDG of the system. The isomorphic subgraphs are computed following the dependence order from any equal node. The clone pair can be extracted from the isomorphic subgraph.

  With control and data dependency information, PDG–based method is the only one that can detect type 3 clone precisely. However, the process is very inefficient, generating PDG, as same as AST parsing process, is a very expensive process on both time and memory, further more finding isomorphic graph is an NP problem.

As clone detection belongs middle section across the fields, some other techniques within string searching , data mining and similar pattern matching field have also been proposed, but most of them are still under the research phrase.

Chapter 3

Problem Statement

3.1 Actual Concerns on Clone Detection

In clone detection field, the main problem we are facing is how to detect all types of clone precisely but consume less time and memory at the same time. As chapter 2 introduced, the source code representation is an essential part of clone detection technique, for good representation will improve not only the accuracy but also the performance in transformation step, comparison step and aggregation step. The main clone detection technique can be classified into two categories, as shown in Table 3.1:

Table 3.1: Classification of clone detection technique

	Text–based	Token–based	Metrics–based	AST–based	PDG–based
Category	textual	textual	textual	semantic	semantic
Supported Clone	type 1	type 1,2	type 1,2	type 1,2,3	type 1,2,3
Complexity	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n^3)$
Meaning of n	line of code	No. of token	No. of method	node of AST	node of PDG

- **Textual level**

  The clone detection technique of this category does not concern any semantic meaning of the source code, so the main advantage is that all of them are very fast, such as, for 100K LOC generally need 100s and 40 MB memory [SR06]. However, without the semantic level information support, only exact copied and pasted clone(type1) and the one with slight modification(type2) can be detected, but the one with further modifications(type3), which is the main disadvantage.

- **Semantic level**

  The clone detection technique of this category needs a complex parsing process to obtain the semantic level information from the source code, so the performances of these techniques are very poor, that is the main disadvantage of these approaches, for instance, processing the same 110K LOC C programmed software SNNS, AST-based tool(CloneDR) needs 3 hours with 628 MB and PDG-based tool(Duplix) needs 63 hours with 64 MB to analyze it[SR06]. However, the advantage of these techniques are that all kinds of clone can be detected, as the additional semantic information can be used to trace the modification after copy and paste of the programmers.

3.2 Good Characters of Clone Detection Technique

From textual to semantic clone detection techniques, the code representation becomes more complex and contains more information which can be used to deeply analyze the source code, although the running time is also increasing. A good Clone detection technique should cope with the trade off between precision and performance at the same time.

The good clone detection should also scale up at least 200K LOC and can also easily adapt to different types of programming languages. The technique should take enough semantic level information from the source code, that not only can be used to detect all types of clone, but also detect more qualitative clones and full clones. The process of collecting semantic information should not be very expensive in time and memory consuming, and the complexity of comparison algorithm should be around $O(n)$. The output of the technique should help programmer maintain the system, for instance, also support outputting clone class.

Chapter 4

Algorithm Design

This chapter describes the design of our clone detection algorithm, the algorithm can be divided into three parts.

Step 1 Performs lexical analysis that detects basic candidate clone pairs

Step 2 Performs dependence analysis that detects further clone pairs based on the candidate clone pairs

Step 3 Groups all clone pairs together into clone class

The nature of this algorithm is, according to the characteristic of the duplicate code, to divide detection process into lexical and dependence analysis two parts. By doing this way, it makes our algorithm have the efficiency of token and string based method, and accuracy of PDG-based method at the same time. The detail of detection algorithm will be described as follow.

4.1 Lexical Analysis

Lexical analysis is the first step, which aims to generate the basic clone pair. Basic clone is the minimum continuous code which constitutes any type of clone. Although, adding and deleting may perform after copying and pasting a code portion, there are still at least some parts of the copied code are not changed. If every line of the copies is changed, then it is not a clone any more. Basic clone pair can only be a type1 or type2 clone, and helps detect type3 clone effectively.

The lexical analysis takes the original source code as input, and takes every statement as a basic unit to analyze. In order to keep performance, a combination of structure code, function code, used data and scope level are computed and recorded for each statement, which is our code representation used to detect clones. As no need to build any complex data structure like AST or PDG, our approach is faster then any other semantic level technique. The lexical analysis can be generally divided into follow steps:

1. Filter uninterested information

   As detecting clone is a very expensive process, especially with dependency analysis involved, all of uninterested information should be filtered before detecting. These information include nonfunctional code which are not actual source code, for example comments or white spaces, and functional code which duplicated on purpose, for instance "include" in C/C++ or "import" in java.

2. Analysis statements

   In order to save time for comparison step, each statement is analyzed token by token and the following information are collected.

   - **Structure code**: a hash code, which is made of serials of token id, represents the structure of each statement. It can be used to generate basic clone pair during suffix comparison step.
   - **Used data**: stores the variable name used in each statement, that helps tracing data flow during the dependence analysis.
   - **Functional code**: stores the control-functional keywords, such as "if", "else", "for", . . . ,etc , that helps tracing control dependence during the dependence analysis.
   - **Scope code**: stores the value and represents level of scope where the statement is, the scope level is increased or decreased by sign "{ " and " }", see table 4.1.

   Table 4.1: Example of scope level

   ```
   scope 0
   {
     scope 1
       {
         scope 2
             {
               scope 3
   }   }     }
   ```

3. Suffix comparison

   To improve the compare speed, a common suffix text searching algorithm is adapted, that with structure code is used as comparison unit. The first step is to categorize the identical structure code of each statement, and then compute the suffixes for each statement, and extract the common suffix branch as basic clone pair at last. The size of basic clone pair can be controlled by the suffixes' size.

4.2 Dependence Analysis

Although some modifications may apply after copy and paste activity, not all of the code will be changed. The unchanged part is already detected as basic clone pair, which may consist of other clone, during suffix analysis. The Dependence analysis takes a basic clone pair, which includes two code fragments as an input. The nature of this algorithm is try to skip the modifications and expand itself into a full clone pair which covered the previous input basic clone pair, then output this full clone pair at last. The figure 4.1 shows the description of this algorithm.

Figure 4.1: Dependence analysis algorithm

Algorithm 1 (Dependence Analysis)

```
Input:  Basic Clone Pair, contains two code fragments $A, B$,
        and each code fragment contains start line and end line
Output: Clone Pair

dependentMatch(BasicClonePair bcp)
1.  while (true)
2.      if (bcp.A or bcp.B is out of their file domain or their scope code $> 0$ )
3.          break
4.      else if (findNextEqual (bcp))
5.          updating current position of A and B
6.      else if (findNextControl (bcp))
7.          updating current position of A and B
8.      else if (findNextData(bcp))
9.          updating current position of A and B
10.     else
11.         break
12.     return ClonePair (bcp)
```

The algorithm 1 needs to perform on both sides, i.e. both start and end position, of the code fragment. The main part of the algorithm is the three get next functions. These functions can adapt to both sides of the code fragment, for instance, for start position of the code fragment, finding next function is to check its previous line, and for end position is to check it's post line.

- **findNextEqual**: The aim of this function is trying to expand the basic clone pair to a full type1 or type2 clone pair. It matches the next line's structure code, and returns true if the structure code is equal, otherwise returns false;
- **findNextControl**: The aim of this function is try to skip the modifications and find the nearest unchanged statement through checking the control flow. It

searches the next several lines within a gap distance, and if it finds any two statements' functional code and scope code are equal returns true otherwise returns false.

- **findNextData**: The aim of this function is as same as the findNextControl function but through checking the data flow instead of the control flow. During the searching step, it returns true if find any two statements' have the same used data and the structure code within a same scope level, otherwise returns false.

The algorithm will be continually running until the basic clone pair can not find any next equal line, any similar control structure line or the line used the same data in the same scope. The algorithm can also terminate when any of code fragment reaches the boundary of the file or reaches the most outer scope.

Table 4.2: Software clone example illustrates algorithm 1

		Code Fragment (a)			**Code Fragment (b)**
1	++	`if (buffer[0] != ‘1’`	1	++	`if (buffer[0] != ‘1’`
2	++	`  && buffer[0] != ‘ ’`	2	++	`  && buffer[0] != ‘ ’`
3	++	`  && buffer[0] != ‘0’`	3	++	`  && buffer[0] != ‘0’`
4	++	`  && buffer[0] != ‘+’) {`	4	++	`  && buffer[0] != ‘+’) {`
5	++	`    if (nread != 1) printf(‘‘\n");`	5	++	`    if (nread != 1) printf(‘‘\n");`
6	++	`    printf(‘‘%s",buffer);`	6	++	`    printf(‘‘%s",buffer);`
7	++	`    nwrite++;`	7	++	`    nwrite++;`
8	++	`};`	8	++	`};`
9	++	`if (buffer[0] == ‘1’)`	9	++	`if (buffer[0] == ‘1’){`
10		`    printf(‘‘\f ");`	10		`    if (nread != 1) printf(‘‘\f ");`
11	++	`if (buffer[0] == ‘‘ ’) {`	11		`  };`
12	++	`   if (nread != 1) printf(‘‘\n");`	12	++	`if (buffer[0] == ‘‘ ’) {`
13			13	++	`   if (nread != 1) printf(‘‘\n");`

Table 4.2 shows an example of type 3 found by our prototype in a real C project(WelTab). As the table shows that the lines marked by ++ are oblivious clone code, and after copy and paste code fragment a, line 10 has been changed. It's a typical type3 clone that programmer keeps the structure and changes the content of the duplicate code. The textual based approach may miss the line 12 and 13 because the gap at 10.

When using our algorithm, for instance, the line 1 to 5 of both code fragments is the input basic clone pair. The basic clone pair will expand to 1 to 9 by findNextEqual function, and stop because the line 10 are not similar. Then by calling findNextControl function, line 11 from code fragment a will match line 12 from code fragment b, because they have the same functional code within same scope level (line 11 from fragment a does not match line 10 from code fragment b, because they are not in same scope). After that the clone pair is continually expanding until line 13 by findNextEqual function.

4.3 Generate Clone Class

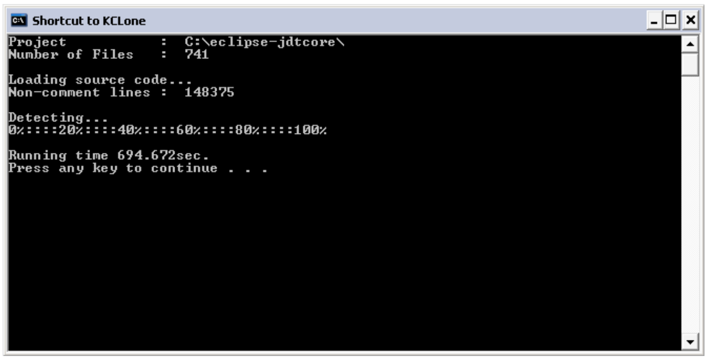
After dependence analysis lots of clone pairs will be proposed, in order to help programmer or system maintainer classify these clones, the clone pair will be grouped together as clone class. There are many ways to group together clone, in this project we use a transformation rule to group clone together, that if code fragment A and B form a clone and code fragment B and C form a clone pair, A , B and C belong to the same clone class.

Chapter 5

Implementation

Our prototype, which named KClone, is a simple program to illustrate our algorithms introduced in chapter 4. Although it currently only supports three programming languages which are C, C++, and Java, new supporting language can be added easily through inserting BNF grammar for the specific language into lexical analyzer.

Our prototype KClone, which developed in C++, is a console based program,see Figure 5.1. It takes two arguments to initialize, one is the directory of the project, the other is the type of the project. KClone will preprocess the files and load all of source code into memory at first, then start detecting clone. During the analysis process, the percentage of the progress will be shown on the screen. After finishing detecting process three files will be saved in the same directory of the project file, one is the project information which records the path , type , and detail source file list of the project, the next stores all clone pairs and the last file stores the clone classes.



```
Project          :  C:\eclipse-jdtcore\
Number of Files  :  741

Loading source code...
Non-comment lines :  148375

Detecting...
0%::::20%::::40%::::60%::::80%::::100%

Running time 694.672sec.
Press any key to continue . . .
```

Figure 5.1: KClone console interface

5.1 System Structure

The kernel part of KClone is KCloneDetector class, which is designed in an object-oriented way, can be easily reused by other projects. The Figure 5.2 shows the class diagram of our prototype.

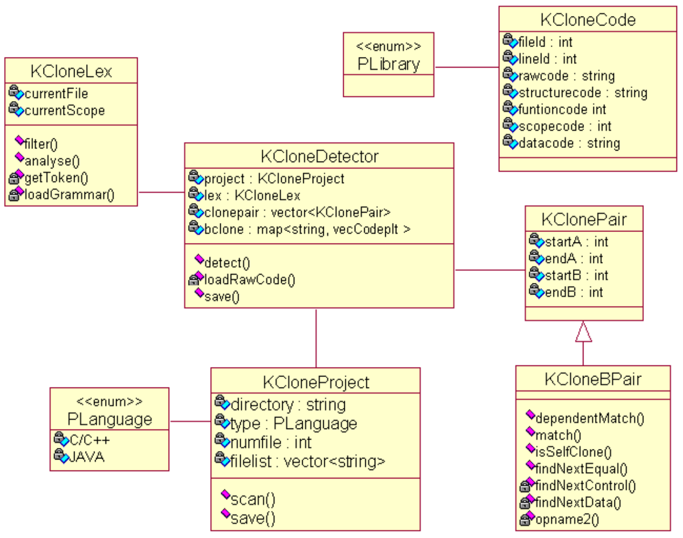


Figure 5.2: KClone class diagram

- **KCloneCode** : Describes the code representation for the source code.
- **KClonePair**: Stores the start and end line position of two similar code fragments.
- **KCloneBPair**: KCloneBPair derives from KClonePair, besides the superclass' data, two functions are added, that match is for finding type1 type2 clone with normal analysis, dependentMatch is for finding all types of clone with dependence analysis.
- **KCloneLex**: KCloneLex is used to analyze the raw code, and all information of code representation is obtained through this process.
- **KCloneProject**: KCloneProject is used to store project information. It takes the path and program language type as an initial argument, then scans the directory and stores the specific source file into file list, according to project type.
- **KCloneDetector**: KCloneDetector takes a KCloneProject object as an initial argument. It loads the raw code according the file list from KCloneProject, and converts to code representation with KCloneLex's help, then performs lexical and dependence analysis to generate clone pair.
- **PLanguage**: Defines the type of supported programming language.
- **PLibrary**: Defines the programming language's grammar.

Figure 5.3 shows the general process of KClone. The program initializes a KClone-Project object by arguments project path and type at first and the KCloneProject will

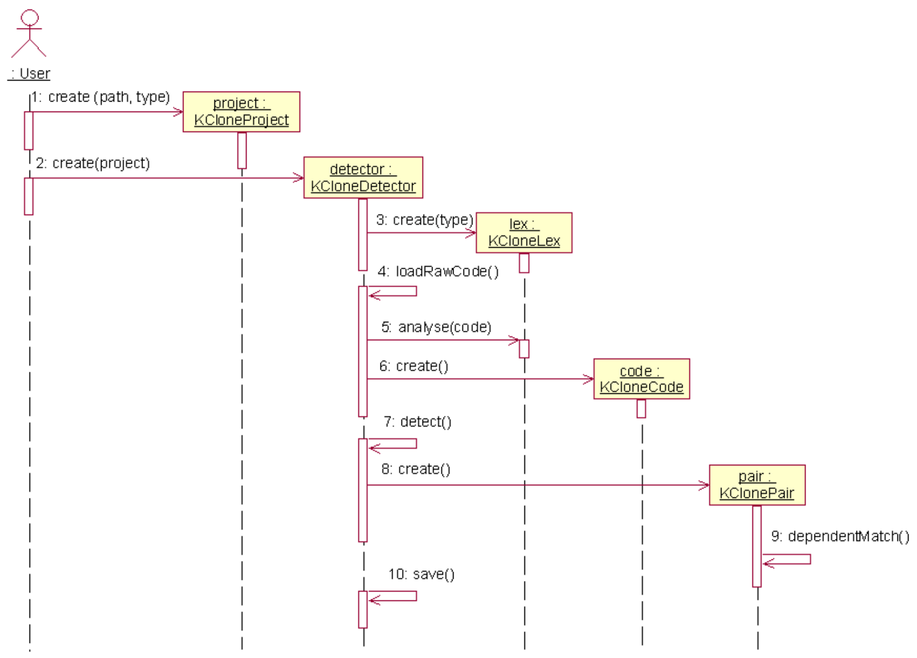


Figure 5.3: KClone sequence diagram

scan the project directory to add relative source file into it's file list. Then a KCloneDetector instance is initialized with the KCloneProject object. The KCloneDetector creates a KClonLex object, and then uses it to analyze each line of code and create code representation KCloneCode, that follows the record of the file list from KCloneProject. After that KCloneDetector performs main detect process, which generates the Basic clone pair at first, and then performs dependence analysis on each basic clone pair as we introduced in chapter 4. After grouping up each clone pair into clone class, the KCloneDetector saves project information, clone pair, and clone class into files at last.

Chapter 6

Experimental Results

Designing a proper experiment to evaluate clone detection techniques is very difficult. The first reason is most of those techniques are running under different environments, which some can run independently without any other support, while others need a specific framework supported, for instance Baxter's CloneDR, an AST–based technique, needs to run under a system named DMS, which provides AST data structure for source code. Secondly, the format of the result may not be same, and the amount of result is also extremely large. Therefore, it's hard to find a way to measure all clone detection techniques at the same time.

Among several clone technique evaluation experiments, the one performed by Stefan Bellon[SR06] in 2002, is the biggest and most completely evaluation for clone detection. Eight systems are picked for the experiment, and six clone detection tools which use different techniques, are chosen, The participators are asked to operate their tool themselves within five weeks, and submit a clone pair file for each project as result, and only the clone that are larger than 5 lines are counted.

At the end of the five weeks, 325,925 clone pairs had been submitted. Considering on performance and other factors, only 2% of clone pairs are selected manually to build a true clone oracle, which is used to measure the precision of techniques according to some metrics.

The advantage of this evaluation way is the true clones which are selected manually, so they can be confirmed that all of them are real clones, however, it is also limited, as only 2% of candidates are covered, so the result may not stand for an average view of all clone detection techniques, e.g. some may happen to find those 2% luckily.

6.1 Experiment Setup

We have run our prototype on the same eight systems and designed an experiment to compare our prototype with other clone detection techniques. Not like Bellon's experiment, we try to cover all of the submitted clone pairs, and focus on the difference between each other, that compares our result with each of them respectively.

Table 6.1: List of clone detector candidates

Participant	Tool	Code Representation
Brenda S.Baker	Dup	Text
Ira D. Baxter	CloneDR	AST
Toshihiro Kamiya	CCFinder	Token
Jens Krinke	Duplix	PDG
Ettore Merlo	CLAN	Function Metrics
Matthias Rieger	Duploc	Text

Besides our prototype KClone, other six clone detection techniques are also selected as candidates (see Table 6.1), we used the result of their detected clone pairs are from the Bellon's experiment, that they submits themselves. Because we want a general evaluation on all of the candidates, and two of the eight proposed testing systems are not supported by at least three other clone detectors, only six systems are selected as test systems(see Table 6.2), which from 11K LOC to 148K LOC.

Table 6.2: List of testing systems

Program	Language	Size
weltab	C	11K LOC
cook	C	80K LOC
snns	C	115K LOC
netbeans–javadoc	JAVA	19K LOC
eclipse–ant	JAVA	35K LOC
eclipse-jdcore	JAVA	148K LOC

Although we intent to study all of the submitted clone pairs manually, there are too many of them in all. In order to quickly find the difference between each tools, we develop a program named KCloneEvaluator. It takes a standard clone pair result file as input, then analyzes each clone pair, if a line of code shows in a clone pair, it will be marked as a clone lines and all clone line and the file will be recorded. The result can be visualized through a clone coverage graph. Figure 6.1 is an example of a clone coverage graph, that x-axis present file id, the y-axis present line id, and the vertical line fragments which are in the middle of the graph, mean that the line fragments from that file belong to at least one clone pair. Each line of clone from any file can be traced back to the original clone pair for further study.

When comparing two clone detection tools through their clone coverage file, we can clearly see which lines of code they propose are both clones, and which lines are found by one but missed by the other. For instance, Figure 6.2 shows an example of a comparison between two clone detection tools. The common clone lines are in black, the blue and red lines indicates that the clone lines are only detected by one tool but missing by the other respectively. After that we can easily take a close look at either their same or different clone lines for further analysis.

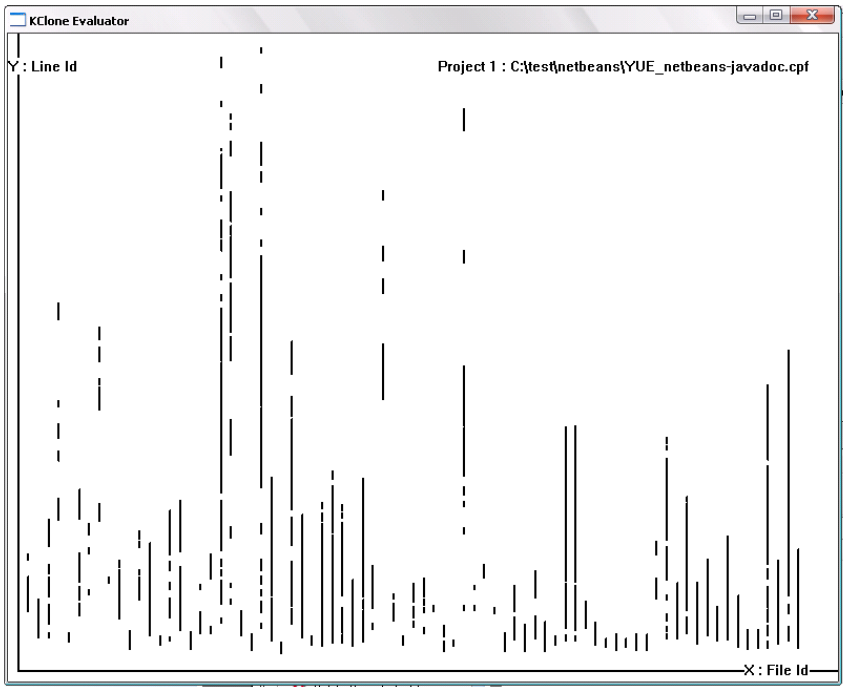
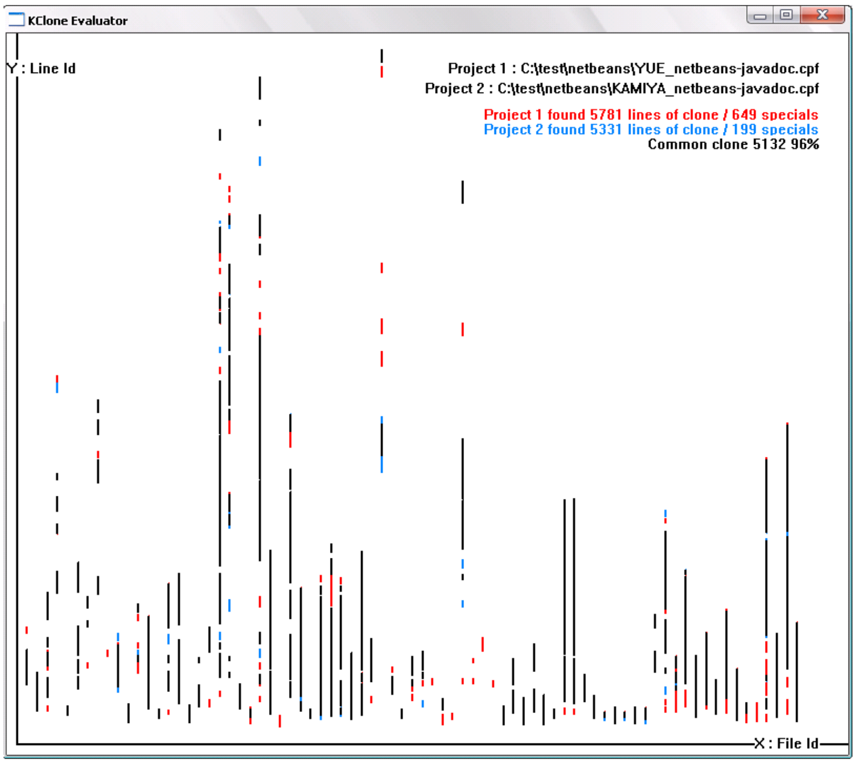


Figure 6.1: Example of clone coverage graph



Project 1 : C:\test\netbeans\YUE_netbeans-javadoc.cpf
Project 2 : C:\test\netbeans\KAMIYA_netbeans-javadoc.cpf
Project 1 found 5781 lines of clone / 649 specials
Project 2 found 5331 lines of clone / 199 specials
Common clone 5132 96%

Figure 6.2: Example of comparison between two detectors through clone coverage file

The experiment starts from generating each detector's clone coverage graph for each testing system, then each of them is going to compare against each other. The following three types of clone line are going to be studied in detail.

- Common clone line: The clone line is found by both detectors.
- Missing clone line: The clone line is found by other detectors, but missing by our prototype.
- Un-proposed clone line: The clone line is found by our prototype but not proposed by others.

6.2 Experiment Results Study

6.2.1 Overall Study

Overall study aims to evaluate the precision and recall of each clone detection tool generally, so we run test on all of six testing systems, and a true clone oracle needs to be built. In our experiment, a union of reported clone lines by at least two detection tools will be selected as a "true clone". The process of generating the true clones starts from comparing every two clone detector's clone coverage file for each project, and finding their common clone lines, and then performs union on the entire common clone lines, and saves the result in true clone oracle at last.

Figure 6.3 lists the overall results for precision experiment for all six systems, and Figures 6.4 shows the average overall results of all of the six systems.For both figures, the first unit of x–axis presents oracle, and the following are detection tools. The y–axis describes the number of clone lines. The blue parts of each detection tools present the number of clone lines, which exists in the oracle, is found by each detectors, and the red parts present the clone lines, which doesn't exists in the oracle but proposed by the detector.

Obviously, it is better that the blue part is as high as the oracle's, and the red is zero for any clone detector. As shown in Figure 6.4, the average overall results shows that KClone's blue part is the nearst to the oracle among all of the detectors, which means it finds most of the true clones, and for some systems(e.g. weltab in Figure 6.3), KClone even equal to oracle's value. However KClone also provides the most un-proposed clone lines which are the red parts. It may seems that KClone makes high false positive, and the reason will explain in section 6.2.3.

6.2.2 Missing Clone Study

During our experiment we found there exists an average 20% of clone lines which KClone can not detect. In order to find how does KClone miss those clone lines, we choose two projects (weltab and javadoc), and compare our result with the other detector to find why we can't detect those clones, and exam the missing clones manually in detail.

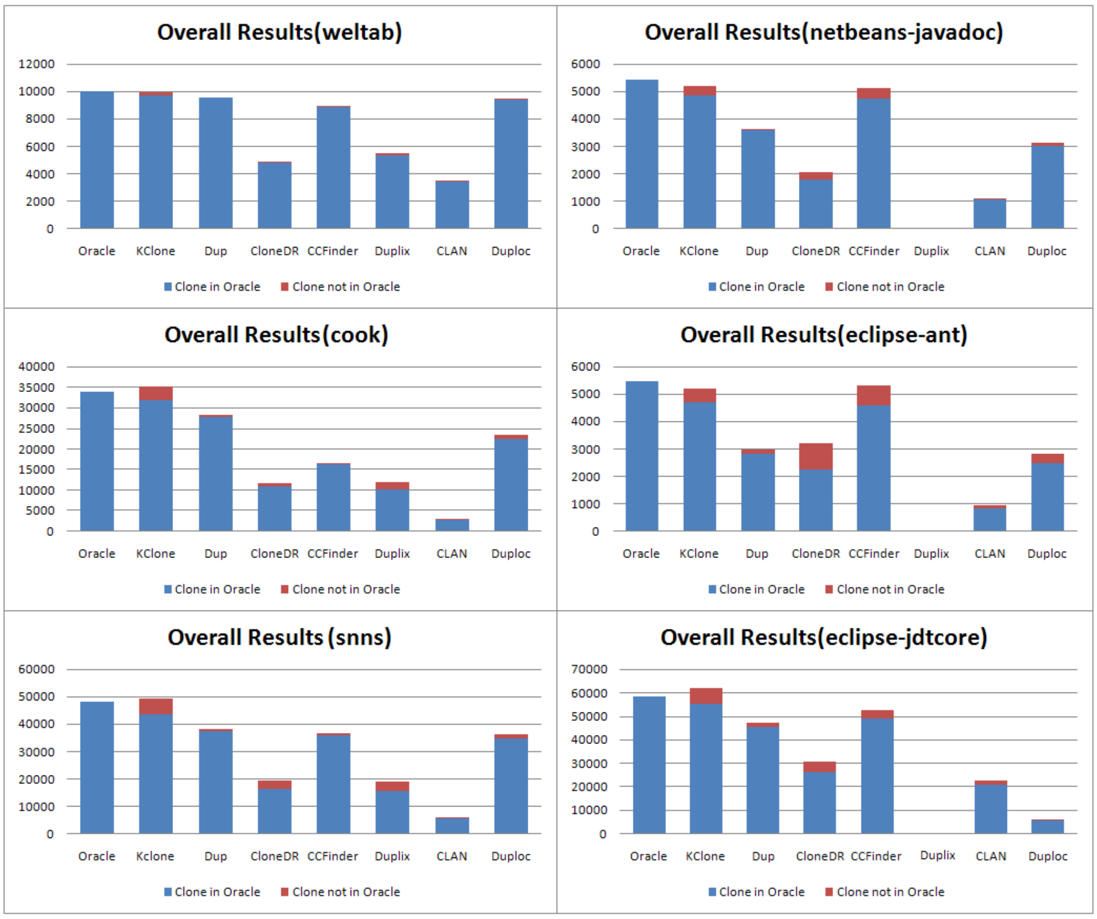


Figure 6.3: Overall results of six testing systems

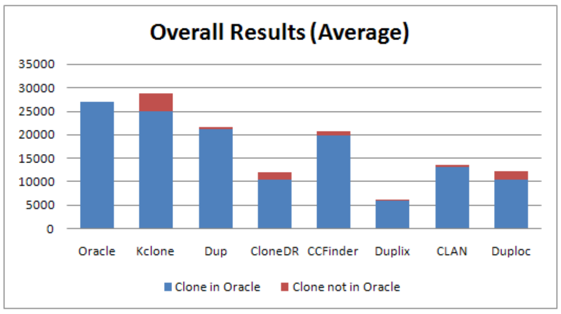


Figure 6.4: Average overall results of six testing systems

Figure 6.5 shows the results of compared KClone's result with each other tools, respectively. The x-axis still presents the detectors and the blue parts show the percentage of clones that found by the detection tool and also found by KClone, and the red parts show the percentage of clones that are found by the detection tool but missed by KClone. As the blues part of each graph are always higher than the 80%, which means that KClone can at least detect 80% clones which are found by any other detectors.

According to this Figure, we trace the clone lines from red parts back the clone pair,

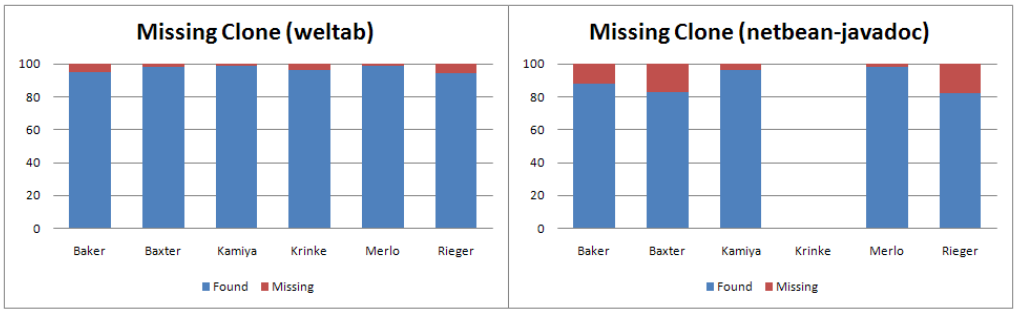
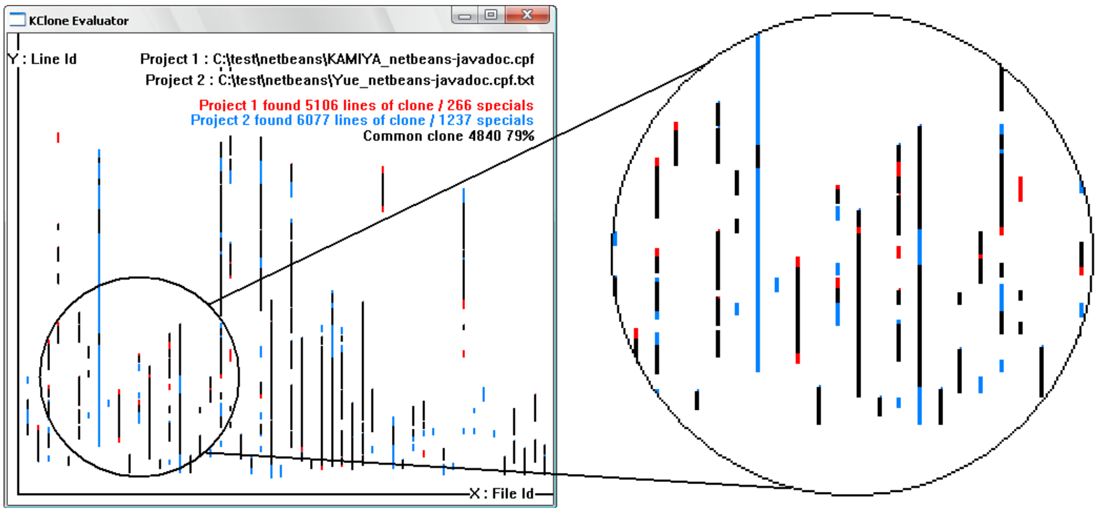


Figure 6.5: Missing clone of weltab and javadoc

then study the reason why KClone miss them. The result can be categorized into three groups:

- **Clone Edge**: The clone edge is the start and end line of the clone, sometimes several clones detectors may find the same clone, but propose different start lines, for instance "}" at the end of the clone pair may be omitted by some tools.

  Figure 6.6 shows the comparison between CCFinder and KClone for netbean-javadoc system. The lines in black present the common clone lines, lines in red present CCFinder found but missed by KClone, and blue lines present the Clone line found by KClone but missed by CCFinder. Form this figure, we can see most of the KClone's missing clone lines(i.e. the line fragments in red), are small dots which connected with a long black line, which indicated the missed clone is a clone edge.

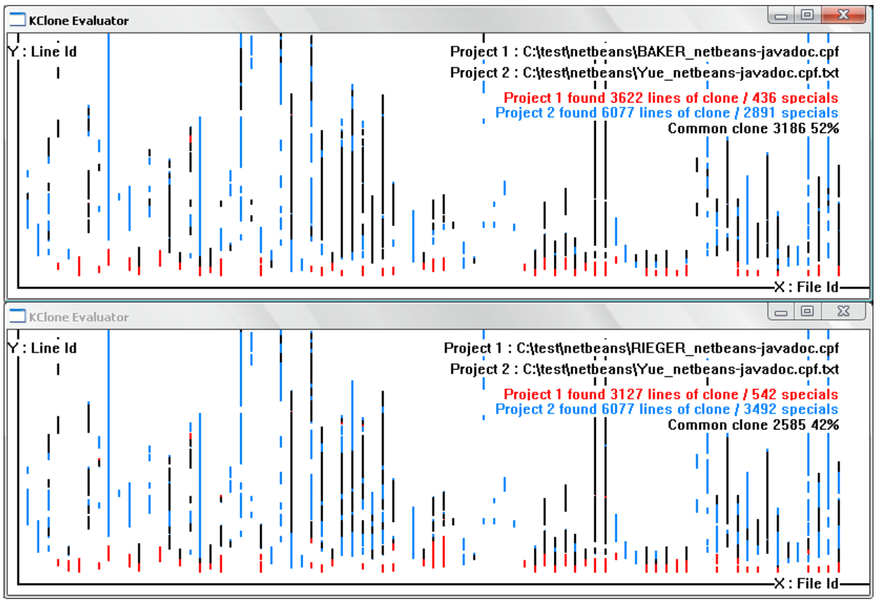


Project 1 : C:\test\netbeans\KAMIYA_netbeans-javadoc.cpf
Project 2 : C:\test\netbeans\Yue_netbeans-javadoc.cpf.txt
Project 1 found 5106 lines of clone / 266 specials
Project 2 found 6077 lines of clone / 1237 specials
Common clone 4840 79%

Figure 6.6: Example of clone edge

- **Uninterested Clone**: When we compare KClone's finding clone with Dup and Duploc for etbean-javadoc system, which an interested pattern of missed clone lines, shows on both of the results, as shown on Figure 6.7. In Figure 6.7, the upside result presents comparison between Dup and KClone, and the red lines show the clone found by Dup but missed by KClone. The downside result shows the comparison between Duploc and Kclone, the red lines show the clone found by Duplic but missed by Kclone. However, we can see all the missing clones are in the end of the graph, which indicates that the missing clones are from beginning of the files.

Then we go through these missing clones one by one and find that most of them are uninterested clones, such as comments, "include" command, "import" command, which are filtered out during the preprocess of our prototype on purpose.



```
Project 1 : C:\test\netbeans\BAKER_netbeans-javadoc.cpf
Project 2 : C:\test\netbeans\Yue_netbeans-javadoc.cpf.txt
Project 1 found 3622 lines of clone / 436 specials
Project 2 found 6077 lines of clone / 2891 specials
Common clone 3186 52%
```

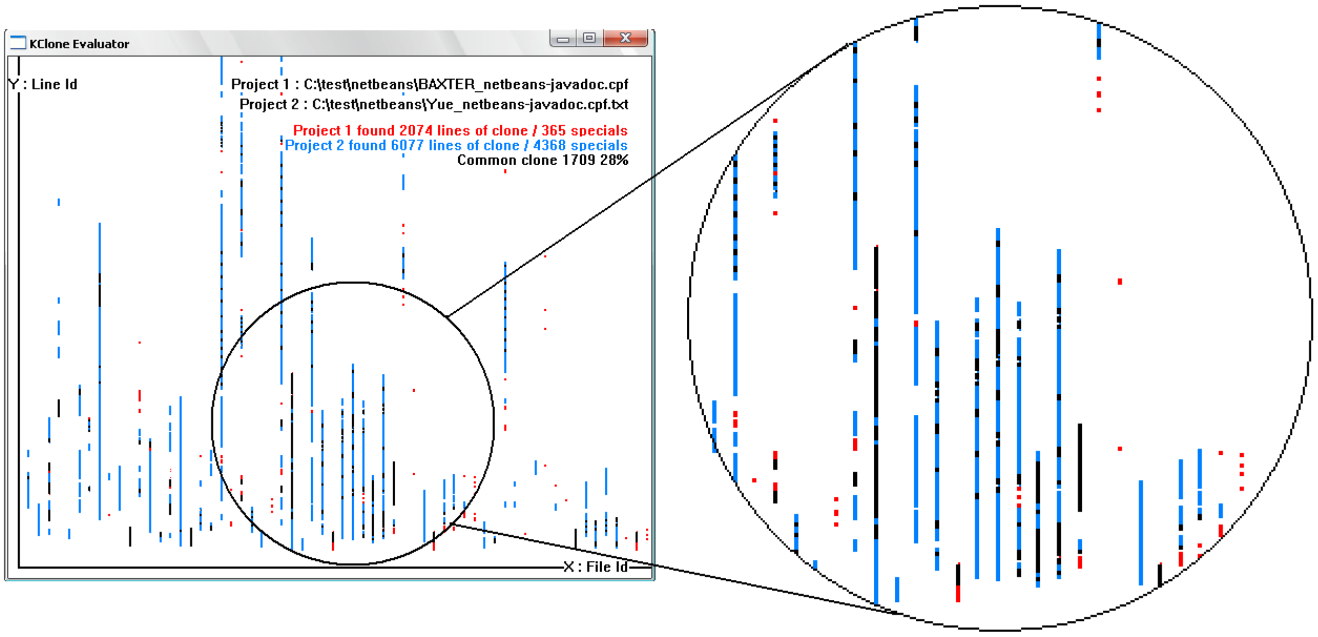
```
Project 1 : C:\test\netbeans\RIEGER_netbeans-javadoc.cpf
Project 2 : C:\test\netbeans\Yue_netbeans-javadoc.cpf.txt
Project 1 found 3127 lines of clone / 542 specials
Project 2 found 6077 lines of clone / 3492 specials
Common clone 2585 42%
```

Figure 6.7: Example of clone edge

- **Tiny clone**: The last group of missed clone is tiny clone, which often consists only two or three lines. Although there is no clear definition for clone size, generally the clone should consist at least five lines, that as same as the rule of this experiment. During our exam of missing clone, we found some detectors proposed a lot of tiny clones in the clone pair result.

  Figure 6.8 shows the compared result between KClone and CloneDR, and the red lines present the clone line found by CloneDR but missed by us. As there is full of the small separate tiny spot which indicates the tiny clones.

As clone edge doesn't affect the precision, and both of the uninterested clones and tiny clones shouldn't exits, we run the text again with ignoring these type of missing clones, as shown in Figure 6.10. The result in Figure 6.10 indicates that our approach can detect around 90% clones which found by any other detectors.



Project 1 : C:\test\netbeans\BAXTER_netbeans-javadoc.cpf
Project 2 : C:\test\netbeans\Yue_netbeans-javadoc.cpf.txt
Project 1 found 2074 lines of clone / 365 specials
Project 2 found 6077 lines of clone / 4368 specials
Common clone 1709 28%

Figure 6.8: Example of tiny clone

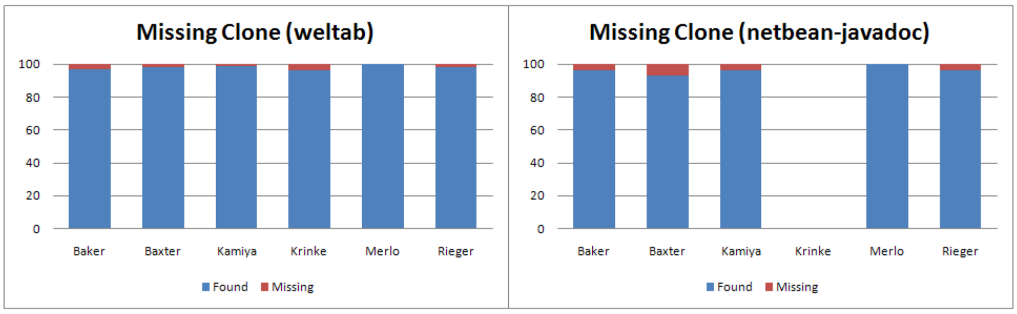


Figure 6.9: Missing clone of weltab and javadoc after filtering

6.2.3 Un–proposed Clone Study

As we mention in overview study, that KClone also detects some clones which not proposed by others, as Figure6.10 shown. In Figure 6.10, the blue parts present the clone detected by KClone and also found by others, the red parts present the clone detected by KClone but missed by others, which we called un-proposed clone.

During our studies, we found most of the un–proposed clones belong to type3 clone. And among those clone detectors, only Duplix is good at detecting this kind of clones. In Fingure6.11, the left graph shows the comparison results between CCFinder and KClone, which black line present the common clone line, and blue lines present the clone found by us but un–proposed by CCFinder. The right graph shows the comparison result between CCFinder and Duploc, which black line presents the common clone line, and blue lines present the clone found by Duploc but missed by CCFinder. The blue line in circle of both graph indicates that the type 3 clone is detected by Duploc which can

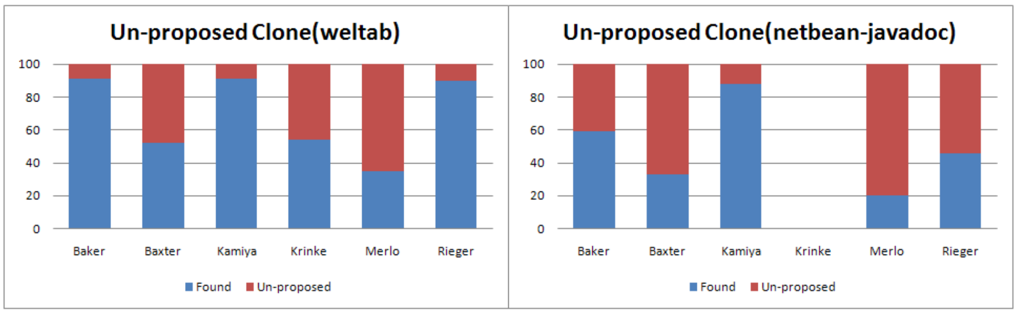
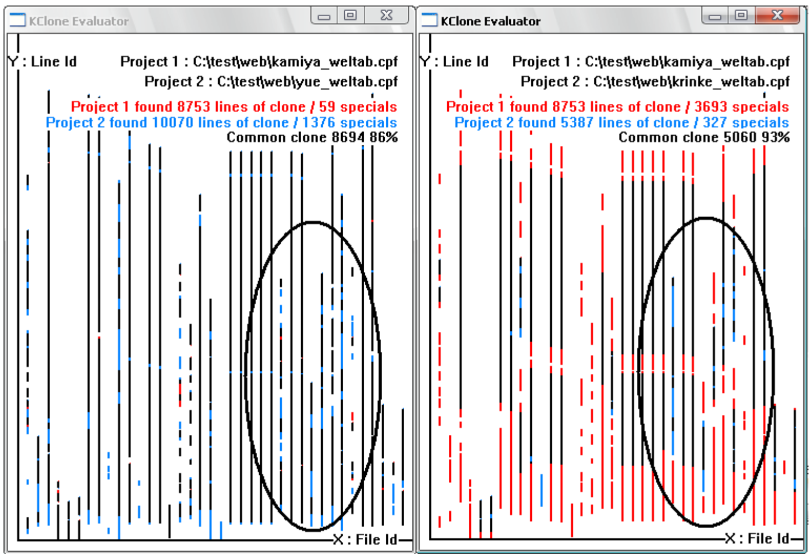


Figure 6.10: Un–proposed clone of weltab and javadoc

also detected by us, but missed by CCFider. But unfortunately, that Duplix can not analyze java programme, so all of the type 3 clones which are detected by KClone for java system will be take into un–proposed clones lead to false positive.



Project 1 : C:\test\web\kamiya_weltab.cpf
Project 2 : C:\test\web\yue_weltab.cpf
Project 1 found 8753 lines of clone / 59 specials
Project 2 found 10070 lines of clone / 1376 specials
Common clone 8694 86%

Project 1 : C:\test\web\kamiya_weltab.cpf
Project 2 : C:\test\web\krinke_weltab.cpf
Project 1 found 8753 lines of clone / 3693 specials
Project 2 found 5387 lines of clone / 327 specials
Common clone 5060 93%

Figure 6.11: Example of un–proposed clone

6.3 Speed and Memory Study

Table6.3 summarizes the running time and memory consuming for our prototype. Because those candidate tools run under different hardware and software conditions, we can't compare the real number with each other. There is some results of worst cases for those tools from Bellon's experiment.[SR06]

The metric–based techniques are the most efficient tool which needs 5 secs to analyze any testing systems. The text–based tool and token–based techniques are also very efficient which generally takes 10 secs with 50 MB memory and 50secs with 50MB memory to analyze around 100K LOC program respectively. However AST and PDG based method are quite slow, the CloneDR(AST) needs 3 hours and 628MB and Duplix(PDG)

Table 6.3: Running time and memory–consuming of KClone

Program	Running time	Memory–consuming
weltab	3s	4MB
cook	422s	24MB
snns	1337s	30MB
netbeans–javadoc	4s	8MB
eclipse–ant	8s	13MB
eclipse-jdcore	713s	22MB

needs 63 hours and 64 MB for SNNS(115 KLOC).

Although our prototype is not the fastest clone detector, but it's obviously, that compare with the semantic type technique, we are far efficient, and almost near to the textual one.

6.4 Summery of Findings

The evaluation result shows that our prototype almost finding 90% clones which are detected by any other tools, and also detecting advanced clones (type 3 clone) efficiently, which previously can only be detected in a long time by some specific tools. It also proves that we have designed a novel clone detection technique which can scale nearly as well as the fast techniques but also can give precision and recall nearly as good as the precise techniques.

Chapter 7

Conclusions

Software clone is a widespread problem in real programs. As a form of reuse, it is usually caused by programmers' copy and paste activities. Although it seems to be a simple and effective method, these duplication activities are usually not documented, that cause a number of negative effects on the quality of the software, increasing the amount of the code which needs to be maintained, and duplication also increases the defect probability and resource requirements.

This project focuses on detecting clone code in a large software system as an aid to maintenance and re-engineering. The novel aspect of this work is that our approach takes the advantage of both textual and semantic type detection techniques, which can detect all kinds of clone effectively. The key benefit of this approach is that it improves both of the precision and performance of clone detection process, for instance, it can detect type 3 clone which normally can only detected by slow semantic clone detection techniques, with similar running time and memory consuming of textual clone detection techniques.

A prototype of our algorithm named KClone has been implemented, which is capable of detecting clones for large software C/C++ system (at least 200K LOC). It can detect all kinds of clone within an acceptable time, and it also supports outputting clone class.

We have also developed an evaluation program, which can be used to generate clone coverage file for any clone detector. By studying the generate clone coverage file for six other industrial clone detection tools on six real testing systems. The result shows our proposed clone detection technique which can scale nearly as well as the fast techniques but also can give precision and recall nearly as good as the precise techniques.

7.1 Future Work

Although we have designed an novel clone detection approach, it is only the first step. The final aim of this project should be automatic clone refactory, which supporting extract clone code into a proper function without programmers' concerning.

To improve our prototype KClone, a GUI interface can be added in order to make it more user-friendly and a more sophisticated algorithm for filtering type 3 clone can also be applied to reduce the potential false positive.

References

[Bak92] Brenda S. Baker. A Program for Identifying Duplicated Code. *Computing Science and Statistics*, 24:49–57, 1992.

[Bak95] Brenda S. Baker. On finding duplication and near-duplication in large software systems. In L. Wills, P. Newcomb, and E. Chikofsky, editors, *Second Working Conference on Reverse Engineering*, pages 86–95, Los Alamitos, California, 1995. IEEE Computer Society Press.

[BMD^{+}99] M. Balazinska, E. Merlo, M. Dagenais, B. Lage, and K. Kontogiannis. Measuring clone based reengineering opportunities, 1999.

[BYM^{+}98] Ira D. Baxter, Andrew Yahin, Leonardo M. De Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377, 1998.

[DRD99] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In Hongji Yang and Lee White, editors, *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pages 109–118. IEEE, 1999.

[Ede93] D. Vera Edelstein. Report on the ieee std 1219c-1993standard for software maintenance. *SIGSOFT Softw. Eng. Notes*, 18(4):94–95, 1993.

[Joh94] J. H. Johnson. Substring matching for clone detection and change tracking. In Hausi A. Müller and Mari Georges, editors, *Proceedings of the International Conference on Software Maintenance (ICSM '94),* (Victoria, B.C.; Sept. 19-23, 1994), pages 120–126, September 1994.

[KDM^{+}96] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. pages 77–108, 1996.

[KH01a] Raghavan Komondoor and Susan Horwitz. Tool demonstration: Finding duplicated code using program dependences. *Lecture Notes in Computer Science*, 2028:383, 2001.

[KH01b] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. *Lecture Notes in Computer Science*, 2126:40, 2001.

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[Kri01] Jens Krinke. Identifying similar code with program dependence graphs. In *Proc. Eigth Working Conference on Reverse Engineering*, pages 301–309, 2001.

[Lei] Leitao Leitao. Detection of redundant code using r2d2.

[LLWY03] Arun Lakhotia, Junwei Li, Andrew Walenstein, and Yun Yang. Towards a clone detection benchmark suite and results archive. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 285, Washington, DC, USA, 2003. IEEE Computer Society.

[LPM$^+$97] Bruno Lague, Daniel Proulx, Jean Mayrand, Ettore M. Merlo, and John Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *ICSM '97: Proceedings of the International Conference on Software Maintenance*, page 314, Washington, DC, USA, 1997. IEEE Computer Society.

[LS80] Bennet P. Lientz and Burton E. Swanson. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley, 1980.

[MM01] Jonathan I. Maletic and Andrian Marcus. Supporting program comprehension using semantic and structural information. In *International Conference on Software Engineering*, pages 103–112, 2001.

[Rie05] Matthias Rieger. *Effective Clone Detection Without Language Barriers*. PhD thesis, University of Berne, 2005.

[SR06] Belln Stefan and Koschkee Rainer. Comparison and evaluation of clone detection tools. *Transactions on Software Engineering*, 2006.

Appendix

Source File list

- kcloneproject.h
- kcloneproject.cpp
- kclonecode.h
- kclonecode.cpp
- kclonelex.h
- kclonelex.cpp
- kclonedetector.h
- kclonedetector.cpp
- kclonepair.h
- kclonepair..cpp
- clonepairprocessor.h
- clonepairprocessor..cpp
- codefragment.h
- codefragment.cpp
- cloneevaluator.h
- cloneevaluator.cpp