Software Clone Detection Based on Context Information

Xianpeng Zhang
Abstract

Software Clone Detection Based on Context Information

Xianpeng Zhang

Software clone detection is very promising and innovative within the industry field. Existing mainstream clone detection techniques mainly focus on detecting the similarity of source code itself, which makes them capable of detecting Type I and Type II clones (Type I clones are two identical code fragments except for variations in format and Type II clones are two structurally identical code fragments except for variations in format). But they rarely pay attention to the relationship between codes. It becomes an important research area to detect Type III code clones, which are clones with minor difference in statements, by using the context information in the source code.

I carry out a detailed analysis of existing software clone detection techniques in this thesis. It raises issues of existing software clone detection techniques in theory and practice. On the basis of the analysis, I propose a new method to improve existing clone detection techniques with a detailed theory analysis and experimental verification. This method makes detection of Type III software clones possible.

Keywords: Software Clone, Context Information, Software Maintenance
Acknowledgement

First and foremost, I would like to thank my supervisor Dr. Liu, Yan. I cannot finish my thesis without your caring and help. I really appreciate your guidance to my thesis and comments on my report.

I also would like to thank my family. It is your financial support and psychologic support that makes me able to study in Sweden, which is an unforgettable experience for me forever.

And I also would like to thank all the people involved in the Sino-Swedish master program, Anders Berglund, Yang Xiaowen and other friends in Sweden or China for providing me a great opportunity to see a different world and to experience a different culture.

At last, I would like to thank my friends and teachers in Tongji University for providing the opportunity to learn from you. The three years of master study is one of the best experiences in my life.
# Table of contents

**Software Clone Detection Based on Context Information** .......................... I

**Abstract** ........................................................................................................ III

**Acknowledgement** ......................................................................................... IV

**Chapter 1 Introduction** .................................................................................. 1

1.1 Structure of thesis ......................................................................................... 1

1.2 Research background ................................................................................... 1

1.3 Contributions ............................................................................................... 2

**Chapter 2 Introduction and analysis of software clone detection** ............... 3

2.1 Terms and definitions ................................................................................. 3

2.1.1 Software clone terminologies .................................................................. 3

2.1.2 Basic Terminology .................................................................................. 7

2.2 Mainstream detection process of software clone ......................................... 8

2.3 Related work .............................................................................................. 13

2.3.1 Software clone detection research ......................................................... 13

**Chapter 3 Evaluation of mainstream clone detection methods** ................. 15

3.1 Mainstream clone detection methods ......................................................... 15

3.1.1 Clone detection based on text ............................................................... 15

3.1.2 Clone detection based on token ............................................................ 16

3.1.3 Clone detection based on tree ............................................................... 17

3.1.4 Clone detection based on program dependency graph .......................... 17

3.2 Drawbacks of mainstream clone detection methods ................................. 18

3.3 CCFinder - clone detection tool based on token ......................................... 18

**Chapter 4 Clone detection method based on context information** ............ 20

4.1 The approach to improve clone detection .................................................. 20

4.1.1 Definition of evaluation value .............................................................. 20

4.1.2 Improvement target .............................................................................. 20

4.2 Introduction of context information ........................................................... 20

4.2.1 Meaning of introduction of context information ................................. 20

4.2.2 Types of context information ............................................................... 20

4.2.3 Acquisition of context information ....................................................... 21

4.3 Definition of context information ............................................................. 21

4.3.1 Physical distance ............................................................................... 21

4.3.2 Positional relationship between code fragments ................................. 21
Chapter 1 Introduction

1.1 Structure of thesis

In Chapter 1, I introduced basic concepts of software clone, development and current research status of software clone detection in China and abroad. I also elaborated background and importance of this research. Structure and main work of the thesis is introduced in this chapter.

In Chapter 2, I briefly talked about terms related with software clone, especially important concepts and terms in general software clone detection process. An overview of papers in the field of software clone detection is shown in Chapter 2.

In Chapter 3, I analyzed mainstream detection methods existing in the field of software clone detection one by one. Advantages and disadvantages of each method are pointed out in this chapter. I also introduced a token-based software clone detection tool called CCFinder.

In Chapter 4, I pointed out why and how to introduce context information during software clone detection. There is also a detailed description of improved software clone detection algorithm.

In Chapter 5, I introduced development environment and the overall design of the detection program. Besides this, I also conducted some experiments and analyzes experiment results. I used experiment results to compare with detection method used in CCFinder with the improved detection method and made some conclusions.

In Chapter 6, I made a conclusion of the thesis, analyzed existing problems of current research and suggested what should be done in the future.

1.2 Research background

Code reuse has been one of the most important topics of software development since the birth of software. To make better use of existing code, it is necessary to analyze existing software source code. Analysis and research on source code has been a hot area in the recent thirty years, and software clone detection and analysis is becoming a popular topic in the area. Software clones (also called as code clones) are code fragments that are same or similar in the software source code. These code fragments may be exactly the same, or slightly different due to minor modification. The modification can either be just editorial, such as changing a variable’s name, or logical, for example, adjusting the structure of the program.

Software clone is not such a big concept as the name suggests, actually, it only focuses on the clones in the sources code. In this thesis, the term ‘software clone’ is the same concept as ‘code clone’. Software clones mainly come from copy-paste code reuse, but also can be a result of patterns or ways of thinking to solve similar problems. Software clone abounds in large software systems. Studies have shown that there are 5% to 20% software clones in the source code of large software systems [2,34]. The ratio of duplicate code is even higher in an object-oriented COBOL system, which is nearly 50% [1]. In most cases, software clone is harmful. Many researches show that software systems with software clones are harder to maintain than those without software clones [2,3]. Software clones make the length of source code in a software system longer, what’s worse, if there is a mistake in one code fragment, then similar or same code fragments probably have the same mistake. All of this becomes a burden to software comprehension and software maintenance. Considering expensive maintenance costs due to the large amount of software clones existing in large software systems, it is of great importance to detect software clones in order to do software refactoring and other tasks related to software maintenance.
Many researchers have done a lot of studies about detection of software clones in recent years. Some of them also propose many effective detection methods [6,7,8]. But many problems are also found during researches [6]. Roy and Cordy list many public problems of software clone detection and this list has been updating since it was created in 2003 [6]. However, the answers to the core issues are still “unresolved” or “partially resolved”. The problem of “How to effectively detect Type III software clone” belongs to the “partially resolved” part.

I proposed introducing context information in source code in this thesis, in order to improve current clone detection methods after a detailed investigation and evaluation of existing mainstream software clone detection. Based on an existing software clone tool (CCFinder), I try to elaborate how to introduce context information in source code in order to improve detection of Type III software clones by realizing this new software clone detection tool.

1.3 Contributions

Evaluate existing software clone detection techniques in details.
I did a detailed evaluation in this thesis to several mainstream techniques in current field of software clone detection and discusses their advantages and disadvantages.

Use context information of source code to detect software clone
I introduced context information of source code to detect software clone in this thesis. I discussed what is context information of a piece of code and why and how to use context information during software clone detection.

Improve existing software clone detection techniques and implement a new software clone detection tool
I improved current software clone detection techniques based on CCFinder. I proposed and implemented a new software clone detection tool. This tool can successfully detect part of Type III software clones with acceptable complexity and a high recall rate and accuracy. This new detection tool is also applied and evaluated in some software systems in this thesis.
Chapter 2 Introduction and analysis of software clone detection

2.1 Terms and definitions

To keep accuracy and consistency of all the terms, I use original terms or explanations in brackets in this thesis.

2.1.1 Software clone terminologies

Code fragment

A code fragment is a sequence of consecutive code lines. A code fragment can contain code of different types and different hierarchies, which is decided by location of the code fragment in the source code. An implementation of a function, a switch statement block and a part of a conditional statement are all examples of code fragments. A code fragment can be represented by three parameters related with its location in the source code: file name (FileName), starting line number (Startline), end line number (Endline). Figure 2.1 shows a code fragment in the source code file of PostgreSQL (FileName: localtime.c, Startline:1382, Endline:1397).

Filename : localtime.c

```c
---
1382  if (t < sp->ats[0])
1383    *boundary -= seconds;
1384  else
1385    *boundary += seconds;
1386  return result;
1387 }
1388
1389  if (t >= sp->ats[sp->timecnt-1])
1390  {
1391    /*No known transition > t, so use last known segment's type*/
1392    i = sp->types[sp->timecnt-1];
1393    ttisp = &sp->tts[i];
1394    *before_gmttoff = ttisp->tt_gmttoff;
1395    *before_isdst = ttisp->tt_isdst;
1396    return 0;
1397  }
```

Figure 2.1 Example of code fragment

Code clone

Code Clone is a code fragment in the source code of software system, which has at least another same or similar code fragment in the source code. Code clones are usually represented as a Clone Pair (two code clones) or a Clone Class (code clones of the same type). Clone Pair and Clone Class are two forms of code clones that don’t conflict with each other, because two code fragments belongs to a same Clone Class is a Clone Pair at the same time.

Code similarity

Code similarity refers to the extent two code fragments are similar. Code similarity can be defined in different ways. It can be defined by comparing text content of code fragments or by comparing...
structure of code fragments. Different code clone methods may use their own definition of code similarity.

**Clone relation**
A Clone Relation is defined as an equivalence relation (this relation is a reflexive, transitive and symmetric relation) between two independent code fragments by a given definition of similarity. The definition of similarity can be defined in different ways. For example, if the given definition of similarity is that two code fragments with no difference exists in text can be treated as similar, then two identical code fragments hold a clone relation under this given definition of similarity.

**Clone class**
A Clone Class is the maximal set of code fragments in which any two of the code fragments hold a Clone Relation. For example, in the code of reading and writing logs of a certain type of system, all identical code fragments form a clone class.

**Clone pair**
A Clone Pair is a pair of code fragments which are identical or similar to each other. In order to facilitate detection and application of clone pairs, extra information is given to clone pairs as their properties, such as size and similarity of Clone Pairs.

**Candidate clone pair**
Candidate clone pair refers to two code fragments which are not processed by any clone detection methods or two code fragments which are in the process of clone detection. A Candidate Clone Pair can either hold a Clone Relation or not. They are objects to be processed.

**Clone type**
There are different Clone Types due to different clone similarity definitions. Basically, there are two similarity definitions, which are based on textual similarity and semantic similarity.

The following are four main types of code clones in the field of software clone.

**Type I code clone**, which is also called Exact Clone, are two identical code fragments except for variations in whitespace, layout and comments. The following is the definition of Type I clone: two code fragments whose code can be identical after text reorganization and normalization. Code fragments illustrated in Table 2.1 is an example of Type I clone.

<table>
<thead>
<tr>
<th>Code Fragment 1 (hello.cpp)</th>
<th>Code Fragment 2 (world.cpp)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>startLineNumber:79</strong></td>
<td><strong>startLineNumber:125</strong></td>
</tr>
<tr>
<td>if (a!=b)</td>
<td>if a is not equal to b, then change a</td>
</tr>
<tr>
<td>{ a=a+b;</td>
<td>if (a!=b){</td>
</tr>
<tr>
<td>c=a;</td>
<td>a=a+b;</td>
</tr>
<tr>
<td>}</td>
<td>c=a;}</td>
</tr>
</tbody>
</table>

Though belonging to two files, the above two code fragments are basically identical except difference in layout and comments. So they are Type I clone.

**Exact clone**
Two or more code fragments which are identical except for variation in whitespace, comments or layout are called Exact Clone. Exact Clones are essentially Type I clone.
**Type II code clone**, which is a superset of Type I clone, are two structurally identical code fragments except for variations in identifiers, literals, types, layout and comments. The following is the definition of Type II clone: two code fragments whose code can be identical after text reorganization and identifier normalization. Code fragments illustrated in Table 2.2 is an example of Type II clone.

<table>
<thead>
<tr>
<th>Code Fragment 1 (hello.cpp)</th>
<th>Code Fragment 2 (world.cpp)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>startLineNumber:100</strong></td>
<td><strong>startLineNumber:125</strong></td>
</tr>
<tr>
<td>static void sig_cid(int signo) /<em>interrupts pause()</em>/</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>pid_t pid;</td>
<td>pid_t pid;</td>
</tr>
<tr>
<td>int status;</td>
<td>int new_status;</td>
</tr>
<tr>
<td>printf(&quot;SIGCLD received\n&quot;);</td>
<td></td>
</tr>
<tr>
<td>if((pid=wait(&amp;status))&lt;0) /<em>fetch child status</em>/</td>
<td></td>
</tr>
<tr>
<td>perror(&quot;wait error&quot;);</td>
<td></td>
</tr>
<tr>
<td>printf(&quot;pid = %d\n&quot;,pid);</td>
<td>printf(&quot;pid = %d\n&quot;,pid);</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

Code fragments in Table 2.2 are identical except for variation in function name and name of two local variables (*status* and *new_status*). These two code fragments can be changed into two identical code fragments if all the variables and functions are renamed by following the same naming convention, for example, renaming all functions by the order they are defined to func1, func2...

**Renamed clone**

Two or more code fragments which are identical to each other except for variation in identifier names, literal values, whitespace, comments or layout are called Renamed Clone. Renamed Clones are essentially Type II clone.

**Type III code clone**, which is a superset of Type II clone, maybe different in some statements. Type III clone is a very common clone type in real life. Code fragments illustrated in Table 2.3 is an example of Type III clone.

Comparing to left code fragment, code fragment on the right removes two printf statements which are for debugging use. Though these two code fragments have some difference in logic, they are clones that should be detected by clone detection techniques from the perspective of application in reality. They have similar functions and are similar in structure, which are valuable software clones.

There is no unified definition of Type III clone. In general, Type III clone is determined by computing the similarity based on some standards and comparing it with a threshold value.

**Near-miss clone**

Near-Miss Clone is exactly the same as Type III clone.

Type IV code clone is two code fragments which have same functionality, but are implemented in different ways. Code fragments illustrated in Table 2.4 have the same function of sending a signal to a subprocess. They are totally different except that they are both functions and have the same parameter list. It is an ultimate goal for software clone detection techniques to find out Type IV clones. But nowadays, there is no perfect solution to detect them.
Table 2.3 Example of Type III code clone

<table>
<thead>
<tr>
<th>Code Fragment 1 (hello.cpp)</th>
<th>Code Fragment 2 (world.cpp)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>startLineNumber:100</code></td>
<td><code>startLineNumber:125</code></td>
</tr>
<tr>
<td>static void sig_cid(int signo) /<em>interrupts pause()</em>/</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>pid_t pid;</td>
<td>pid_t pid;</td>
</tr>
<tr>
<td>int status;</td>
<td>int status;</td>
</tr>
<tr>
<td>printf(&quot;SIGCLD received\n&quot;);</td>
<td></td>
</tr>
<tr>
<td>if((pid=wait(&amp;status))&lt;0) /<em>fetch child status</em>/</td>
<td></td>
</tr>
<tr>
<td>perror(&quot;wait error&quot;);</td>
<td></td>
</tr>
<tr>
<td>printf(&quot;pid = %d\n&quot;,pid);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

Table 2.4 Example of Type IV code clone

<table>
<thead>
<tr>
<th>Code Fragment 1 (hello.cpp)</th>
<th>Code Fragment 2 (world.cpp)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>startLineNumber:125</code></td>
<td><code>startLineNumber:90</code></td>
</tr>
<tr>
<td>static void sig_cid(int signo) /<em>interrupts pause()</em>/</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>pid_t pid;</td>
<td>do_sig_cld(signo);</td>
</tr>
<tr>
<td>int status;</td>
<td>}</td>
</tr>
<tr>
<td>printf(&quot;SIGCLD received\n&quot;);</td>
<td></td>
</tr>
<tr>
<td>if(signal(SIGCLD,sig_cld)==SIG_ERR)</td>
<td></td>
</tr>
<tr>
<td>perror(&quot;signal error&quot;);</td>
<td></td>
</tr>
<tr>
<td>if((pid=wait(&amp;status))&lt;0) /<em>fetch child status</em>/</td>
<td></td>
</tr>
<tr>
<td>perror(&quot;wait error&quot;);</td>
<td></td>
</tr>
<tr>
<td>printf(&quot;pid = %d\n&quot;,pid);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

Type I, Type II and Type III clones are software clones based on textual similarity and Type IV clone is software clone based on semantic similarity. It is noteworthy that definitions of the above four types of clones are just a broad definitions widely accepted by many researchers in the field [7,8], and they are not strict formalized definitions. I make some supplement in the following section.

Reordered clone
Reordered Clone is a special type of code clone. It refers to two or more code fragments whose control flow is similar. A reordering of some segments may be possible in the copied fragment that do not alter the data or control dependencies of this fragment comparing to the original. Reordered clone could be Type III or Type IV clone.
In the above example, even though some statements are ordered and variables are renamed, the functionality of code fragment 1 is the same as code fragment 2.

**Parameterized clone**

Parameterized clone is a special kind of Renamed Clone. A parameterized clone is a renamed clone with systematic renaming.

Code fragments illustrated in Table 2.6 show an example of Parameterized Clone. Code fragment 1 can be changed to be the same as code fragment 2 if renaming \( a \) to \( i \) and \( b \) to \( j \) consistently. So code fragment 1 and 2 are a pair of parameterized clone. We can use a template with parameters to represent the entire class of code. For example, code in fragment 2 can be used as a template, while code fragment 1 is an result of parameter replacement based on some fixed rules using this template. Code fragment 3 is not a Parameterized Clone with code fragment 1 or code fragment 2 since parameter replacement is not done consistently.

### Table 2.6 Example of Parameterized Clone

<table>
<thead>
<tr>
<th>Code Fragment 1</th>
<th>Code Fragment 2</th>
<th>Code Fragment 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>if(a&lt;b)</td>
<td>if(i&lt;j)</td>
<td>if(i&lt;j)</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>b--;</td>
<td>j--;</td>
<td>i--;</td>
</tr>
<tr>
<td>a=4;</td>
<td>i=4;</td>
<td>j=4;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

#### 2.1.2 Basic Terminology

Some terminologies related with code or program are listed in Table 2.7 with their meaning.
2.2 Mainstream detection process of software clone

A lot of methods of detecting software clones have been created since Yang [4] firstly came up with the concept of software clone detection at 1991. Though all of the detection methods are different, most of them follow a general process procedure. The general process procedure can be presented by Diagram 2.2:

Table 2.7 Basic terminologies used in clone detection

<table>
<thead>
<tr>
<th>Terminology</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Code</td>
<td>Compilable part of program text</td>
</tr>
<tr>
<td>Token</td>
<td>A joint name for variable identifier, class type name and keyword.</td>
</tr>
<tr>
<td>Token Stream</td>
<td>A token sequence derived from grammar analysis of source code</td>
</tr>
<tr>
<td>Abstract Syntax Tree</td>
<td>A type of parse tree derived from syntax parse result of token sequence</td>
</tr>
</tbody>
</table>

The following is the general process of clone detection:

**Preprocessing**

In the process of preprocessing, code fragments are filtered and processed into some form of code to be used in next steps. Procedures are adopted depending on implementation of detection
methods and application requirement. The following part will deliver a brief introduction to these preprocessing procedures.

1. **Normalization**

Normalization filters unnecessary information without affecting function of the program in order to eliminate redundant part that may affect the detection result in the code before comparison. Normalization includes removing whitespace, comments and other unrelated information.

   • **Removing comments**
   Most detection methods disregard comments because comments are not involved in implementation of code logic. But it is undeniable that comments contains a lot of meaningful information, and it should be considered to keep information contained in comments if we can make best use of it.

   • **Removing whitespace**
   Almost all approaches disregard whitespace and many methods remove extra line breaks. However, line-based approaches probably keep all the line-breaks and some metric-based methods take layout into account as well. For programming languages, such as Python, to whom indents and whitespace is grammatically meaningful, whitespace and indents should be kept until the period of lexical analysis. It is the same for Tab symbols in front section of a makefile.

   • **Removing irrelevant part**
   Code text contains lots of information both in type and quantity. The information is probably overmuch for one specific detection. So it is necessary to select appropriate information used in detection and remove information that may interfere with process phase before doing clone detection. The irrelevant parts can be but are not limited to package import declaration (used in Java language), auto-generated code fragments and other information that is not interested by the clone detector.

   • **Normalizing layout**
   Normalizing layout means normalizing code by some fixed format, which is a simple but effective way to deal with slight difference in text caused by different layouts. Normalizing layout transforms similar code to code with exactly the same format and layout. Only by this way, Type I and Type II clone pairs caused by extra blank line and different code layout can be detected. This operation is based on lexical analysis, so it is not as effective as previous operations. The loss may outweighs the gain when this operation is applied in large-scale system, so it is also optional.

   • **Normalizing identifiers**
   Most of the approaches apply identifiers normalization before the comparison phase in order to detect Type II clone. All identifiers of the source are replaced by a single token in such normalizations. After identifiers normalization, Type II clone pairs are transformed to Type I clone pairs. Identifier normalization removes the order information which has a big effect on detection of Type III clone detection, so identifier normalization should be reserved except for independent detection process.

2. **Filtering**

Filtering phase deletes unnecessary part for detection from identifiers and keywords. Filtering usually is done with normalization phase mentioned in the previous section.

**Code Analysis**

The goal of code analysis is to transform preprocessed code to to-be-processed objects which are comparable to each other. Following comparison operation will be done on the to-be-processed
objects. Code analysis mainly includes grammar analysis and syntax analysis. Code is firstly transformed into token sequences composed of grammar symbols and then parsed into a syntax tree based on the language’s syntax during the phase of code analysis.

1. **Grammar analysis**

Grammar analysis mainly refers to tokenization. This operation is a key point for detection methods based on token. Even if for methods based on **AST (Abstract Syntax Tree)** or methods based on **PDG (Program Dependency Graph)**, grammar analysis is one of the required steps.

Each line of code is divided into a series of tokens based on the grammar rule of programming language and all tokens in a code fragment are transformed into a token stream after tokenization. Most of the clone detection methods also perform a series of normalization operation during this phase.

Because of the particularity of program codes, it is often desirable to represent codes as a token stream as illustrated in Diagram 2.4. Tokens referred here are tokens in lexical meaning. In Diagram 2.4, an simple *if* statement in C programming language is transformed into to a token stream. Every token in this token stream is a stationary structure of lexical specification, such as ‘if’, ‘(’, ‘{’, ‘a’ and so on. Syntax parsing transforms the token stream into an abstract syntax tree or a parse tree using syntax specification of the programming language.

![Diagram 2.4 Example of token stream](image)

2. **Syntax analysis**

Syntax analysis is not a required step for detection methods based on text or token. But it is of great importance for methods based on syntax parse tree. Syntax analysis parses the whole code file to an abstract syntax tree or a syntax parse tree. Every leaf node of the tree is a token and every subtree represents an abstract syntax unit. An example is shown in Figure 2.5.
It can be seen from the diagram that every syntax unit in the program, such as expression, function and so on, is corresponded with a subtree of a parse tree or abstract syntax tree. Detection methods based on syntax tree use subtree-comparison algorithm to compare abstract syntax tree of two code fragments to look for potential clone pairs.

Figure 2.5 shows an abstract syntax tree used by DECKARD [5]'s detection method. It is an abstract syntax tree for a simple for loop. Root node indicates that it is a for_statement, which is an abstract syntax structure. Every leaf node is a token, such as for, int, id, =, and so on, while every non-leaf node is an abstract syntax structure. So each subtree rooted by any non-leaf node is a complete structure. For example, inc_e node in the figure represents a complete increment statement.

**3. Control and data-flow analysis**

Many methods use PDG technique in order to detect semantic software clones. These methods generate a PDG by doing static analysis on the program codes. Each line of code constitutes a node in the PDG, and this node also represents control flow of the program at some point and its corresponding condition, therefore each edge in the diagram represents the direction of control flow and dependency of the data-flow at the same time. Each syntax unit of the program corresponds to a subgraph in the PDG. Code fragments similar in structure or semantic should have similar PDG. Detection methods can detect similar code fragments by looking for isomorphic subgraphs in the PDG or deduce similarity of two code fragments through isomorphic situation of subgraph of the code fragments.

Detection methods based on PDG generate PDG when processing source code and many methods based on metrics also need to generate PDG to get metrics of some data and control flow. Figure 2.6 shows a PDG of a simple function. This function has two int-type value input and defines a variable k. Two paths can be found in the figure. One of them is data-flow represented by the solid line, which shows how the function deals with the input parameters and user variables and the other is control-flow represented by the dotted line, which shows logic of program.
4. Calculating metrics
Many detection methods based on metrics calculates several metrics from the raw source code and use these metrics values for finding software clones. Metrics-based clone detection methods usually compute an attribute vector for each comparison unit. For example, Mayrand et al. [30] use metrics to find "an exact copy or a mutant of another function in the system".

Match Detection
This phase is the core part of software clone detection. Most clone detection methods are classified by the operation of this phase and objects processed in this phase. They are categorized as the following:

1. Match detection based on text
To-be-processed objects, which are results of the previous phase of clone detection based on text, are some type of deformation of code text. Match detection phase uses these objects to find out software clones. These methods mainly compare text similarity of the to-be-processed objects to determine whether two code fragments are clone pairs.

2. Match detection based on token
Technique based on token transforms code to token sequences and determines similarity of two code fragments through similarity of token sequences of the two code fragments. Because grammar is not involved in these methods, token sequences actually can be across different syntax units. Clone pairs found this way may have a clone relationship, but have little significance for software refactoring. Therefore, many methods use some type of syntax-related
techniques (or split code block by syntax unit before the match detection phase) to filter clones found in this way.

3. Match detection based on abstract syntax tree

Syntax related methods parse program code to a parse tree or an abstract syntax tree (mainstream methods normally use abstract syntax tree). To-be-processed objects of this phase are two abstract syntax trees generated by software code fragments to be detected. Methods use similarity of structure of subtrees to determine the similarity of original code and find clone pairs. The core concept of this method is widely referenced and used. Methods using this core concept focus on looking for similar subtrees, and the main difference between these methods is the way to find similar subtrees.

4. Match detection based on program dependency graph

Software clone detection technique based on program dependency graph is similar to method based on abstract syntax tree. It generates high-level syntactic structures by doing syntax analysis on code and determines similarity of original code by comparing similarity of syntactic structures. To-be-processed objects of this phase is program dependency graph generated by token sequences.

But there is still no practical software clone detection method based on program dependency graph applied to large-scale software system by the time of finalizing this thesis.

5. Match detection based on metrics

To-be-processed objects of detection methods based on metrics are neither source files nor syntactic structures. They analyze source files and get corresponding metrics when doing match detection. And some calculation are made on these metrics to decide whether code fragments are similar or not or which type of clone these code fragments belong to.

Analyze and Process

Most clone detection methods export matching result of detection in some format after matching detection is done. The result is mainly used for data analysis and evaluation. For methods that are end-user oriented, original source code fragments also should be displayed.

Most software clone detection tools provide a detailed report in the end, in which amount, type and specific location of clones contained in the software system are recorded.

At this point, a common software clone detection process is completed.

2.3 Related work

2.3.1 Software clone detection research

Some of the early studies show that system with more software clones is harder to maintain [2,3] than those with fewer software clones. Current studies have shown that the ratio of software clones in existing systems is much higher than we expect [6,35].

Research of software clone definition

Kamiya [8] offers a vague definition when raising his detection method. He thinks software clone as different parts of code which are ‘identical’ or ‘similar’. Baxter[32] defines software clone as a pair of code fragments which are determined as similar according to a similarity definition. He raises a similarity definition based on threshold value applied for detection of Near-Miss clone. However, he
does not provide an independent similarity definition. Burd [33]'s definition is very similar with Kamiya’s, which is also a mainstream form of software clone definition in early times. Other software clone researchers try to make an independent similarity definition [9,10,11], but progress in this aspect is not significant.

To avoid using the vague definition of ‘similar’, many researchers come up with the use of classification approach to define a software clone. Maryland [30] is a pioneer of this aspect. She tries to provide an explicit determination condition for existing clone detection methods through a series of clone definitions. On the basis of this, Balazinska [20,21,22] etc. comes up with 18 different types of clone definitions and they take syntactic elements into account.

Many researchers also think manual software clone detection is an important research field [7,8]. Their experiment results show that most of the clone candidates detected automatically cannot pass artificial identification, which is another proof that similarity judgement for code clone is still a very difficult problem.

The size of target clones is an important parameter closely related with similarity, which is of great importance when looking for possible clone candidates in software system. Researchers attach great importance to this issue and make some achievements. For example, Robillard [14], Antoniol [15], Aversano [16] all point out that 30 tokens is a reasonable number of minimum threshold value for clone detection techniques based on tokens. But some other researchers have a different opinion, Baker [17], Wahler [18] think that it is more meaningful to use total lines of code to determine the size of target clones. Researchers use AST or PDG naturally choose the size of subtree (AST) or subgraph (PDG) as the threshold value for clone detection. While for researchers who are only interested in function clones, Balazinska, Merlo [21, 22], they limit the threshold value to size of a function.

No matter how different the definition of similarity is, it is undisputed that definition of similarity can be roughly divided into textual similarity and functional similarity. The former can be found in the thesis of Balint [23], Baker [24], Basit [25]. The latter is also called as semantic similarity and can be found in thesis of Basit [26,27,28,29].

**Clone detection technique**

Current clone detection technique is mainly divided into four categories:
1)Clone detection technique based on text
2)Clone detection technique based on token
3)Clone detection technique based on tree
4)Clone detection technique based on program dependency graph

A detailed introduction of these techniques is given in Chapter 3.
Chapter 3 Evaluation of mainstream clone detection methods

To improve existing clone detection methods, it is necessary to evaluate and analyze the detection results of currently existing methods, especially for methods whose target is different types of software clones. I choose methods that are considered with good detection accuracy to present specific type of clone detection method on the basis of thesis and documents of current mainstream detection methods. A detailed and correct analysis is the goal of this section.

3.1 Mainstream clone detection methods

Categories of current mainstream software clone detection methods have been introduced in the ‘related work’ section of chapter 2. This section carries out a further exploration and analysis of the methods and corresponding tools that are used.

3.1.1 Clone detection based on text

On the surface, detection accuracy of methods based on text might not be good, but the actual situation is not the case. This is because all the clone pairs of existing code are often, to a large extent, a result of copy-paste action by programmers while they are coding. This makes methods based on text very useful for clone detection. What’s more, algorithm of the methods is comparatively simple and the computing complexity is the lowest ($O(n)$), and this is why research of this regard has not been terminated. One of the best research results is Nicad, which applies Line-to-Line comparison method in analysis of source code and uses algorithm of Longest Common Substring (LCS) to determine similarity of two code fragments - a basic algorithm model of the diff program in Unix. This section chooses clone methods of Baker [2.17], Johnson [3] and Nicad [7] as representatives for clone detection methods based on text. Table 3.1 makes a brief summary of these three methods.

<table>
<thead>
<tr>
<th>Item to be compared</th>
<th>Baker’s method</th>
<th>Johnson’s method</th>
<th>Nicad’s method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalization or...</td>
<td>Removing space and comments</td>
<td>Removing space and comments</td>
<td>Removing space and comments</td>
</tr>
<tr>
<td>Code representation</td>
<td>Modified token string</td>
<td>Fingerprint of substring</td>
<td>Transformed code</td>
</tr>
<tr>
<td>Comparison technique/...</td>
<td>Suffix-tree based token matching</td>
<td>Karp-Rabin string matching</td>
<td>Longest common subsequence algorithm</td>
</tr>
<tr>
<td>Time complexity</td>
<td>$O(m+n) \quad m$ is number of matches found and $n$ is lines of input code</td>
<td>Not available</td>
<td>$O(n), \quad n$ is the number of lines of input code</td>
</tr>
<tr>
<td>Comparison granularity</td>
<td>Code line</td>
<td>Substring</td>
<td>Code line</td>
</tr>
<tr>
<td>Clone granularity</td>
<td>Free</td>
<td>Free</td>
<td>Function and code block</td>
</tr>
<tr>
<td>Language dependency</td>
<td>Lexer is needed</td>
<td>No lexer/parser is needed</td>
<td>Lexer is needed and corresponding rules of the programming language</td>
</tr>
</tbody>
</table>

Table 3.1 comparison of methods based on text
3.1.2 Clone detection based on token

This type of technique is researched the furthest and most widely in the field. The main reason of this situation is the existence of the excellent code clone detection tool - *CCFinder*. The algorithm of this tool is efficient and mature and can be applied in large-scale system, which makes it popular in reality. *CCFinder* is chosen as a representative tool of software clone detection in many researches and analysis related with software clone detection. Quick detection speed and good effect of traversing large-scale software system is the advantage of this tool. Because clone detection method raised in this thesis is also based on *CCFinder*, section 3.3 will give a detailed introduction of *CCFinder* and its detection algorithm. Besides *CCFinder*, Baker [2] and Li [19]’s methods are also representative methods in this field.

<table>
<thead>
<tr>
<th>Item to be compared</th>
<th>Baker’s method</th>
<th>Johnson’s method</th>
<th>Nicad’s method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output type</td>
<td>Clone pair and clone class</td>
<td>Clone pair</td>
<td>Clone pair and clone class</td>
</tr>
<tr>
<td>Code refactoring</td>
<td>Not needed</td>
<td>Not needed</td>
<td>Not needed</td>
</tr>
</tbody>
</table>

Table 3.2 comparison of methods based on token

<table>
<thead>
<tr>
<th>Item to be compared</th>
<th>CCFinder</th>
<th>Baker’s method</th>
<th>Li’s method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalization or transformation of code</td>
<td>Removing space and comments, transformation and replacement for some parameters</td>
<td>Removing space and comments</td>
<td>Mapping source to collections of sequence with similar statements/ identifiers to the same value token</td>
</tr>
<tr>
<td>Code representation</td>
<td>Sequence of normalized, transformed and parameterized tokens</td>
<td>Parameterized token string</td>
<td>Collection of sequences</td>
</tr>
<tr>
<td>Comparison technique/ algorithm</td>
<td>Suffix-tree based token matching</td>
<td>Suffix-tree based token matching</td>
<td>Frequent subsequence mining technique</td>
</tr>
<tr>
<td>Time complexity</td>
<td>$O(n)$, $n$ is the length of source file</td>
<td>$O(m+n)$, $m$ is number of matches found and $n$ is the number of input lines</td>
<td>$O(n^2)$, $n$ is the number of lines of code</td>
</tr>
<tr>
<td>Comparison granularity</td>
<td>Token</td>
<td>Code line</td>
<td>Sequence of tokens of basic block</td>
</tr>
<tr>
<td>Clone granularity</td>
<td>Free, threshold-based of tokens</td>
<td>Free, threshold based</td>
<td>Free, threshold-based (Basic blocks and functions)</td>
</tr>
<tr>
<td>Language dependency</td>
<td>A lexer is needed and transformation rules for the language</td>
<td>At most needs a lexer</td>
<td>Needs a full parser</td>
</tr>
<tr>
<td>Output type</td>
<td>Clone pairs</td>
<td>Clone pairs and clone class</td>
<td>Clone pairs and clone class</td>
</tr>
<tr>
<td>Code refactoring</td>
<td>Not needed</td>
<td>Not needed</td>
<td>Not needed</td>
</tr>
</tbody>
</table>
3.1.3 Clone detection based on tree

Clone detection methods based on abstract syntax tree is a detection method related to program syntax. The common practice is using a parser to parse program code to a parse tree or an abstract syntax tree (most of the methods parse code to an abstract syntax tree). After the parsing phase, to-be-processed objects are two abstract syntax trees generated from code fragments of software source code being detected. The definition of similarity is based on similarity of subtree structure. These methods looks for similar part of source code based on the definition of similarity. The main difference of different methods based on abstract syntax trees lies in the way looking for similar subtrees. Yang [4], Wahler [18] and Baxter [31]’s methods are representatives of this field. Table 3.3 makes a summary of these three methods.

Table 3.3 comparison of methods based on tree

<table>
<thead>
<tr>
<th>Item to be compared</th>
<th>Yang’s method</th>
<th>Wahler’s method</th>
<th>Baxter’s method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalization or transformation of code</td>
<td>To a variant of parse tree</td>
<td>Parsed to AST and then AST in XML</td>
<td>Parsed to AST</td>
</tr>
<tr>
<td>Code representation</td>
<td>Parse tree</td>
<td>AST</td>
<td>AST</td>
</tr>
<tr>
<td>Comparison technique/algorithm</td>
<td>Tree matching with dynamic programming scheme</td>
<td>Frequent Itemset</td>
<td>Tree matching</td>
</tr>
<tr>
<td>Time complexity</td>
<td>$O(m*n)$, $m$ and $n$ is node number of two parse trees</td>
<td>$O(kn^2)$, $k$ is maximal size clones and $n$ is number of statements containing clones</td>
<td>$O(n)$, $n$ is node number of abstract syntax tree</td>
</tr>
<tr>
<td>Comparison granularity</td>
<td>Token</td>
<td>Code line</td>
<td>A node of abstract syntax tree</td>
</tr>
<tr>
<td>Clone granularity</td>
<td>Free</td>
<td>Free, usually 5 statements, threshold-based</td>
<td>Free, tree similarity based</td>
</tr>
<tr>
<td>Language dependency</td>
<td>Needs a parser and pretty-printer</td>
<td>Maybe need a parser</td>
<td>At least needs a lexer</td>
</tr>
<tr>
<td>Output type</td>
<td>Just displays with pretty-printing</td>
<td>Clone Pair</td>
<td>Not available</td>
</tr>
<tr>
<td>Code refactoring</td>
<td>Not needed</td>
<td>Not needed</td>
<td>Not needed</td>
</tr>
</tbody>
</table>

3.1.4 Clone detection based on program dependency graph

Theoretically speaking, clone detection method based on program dependency graph is the most promising method, because this type of method takes logic of program into account and logically analyzes information of program's control-flow so that it is able to make equivalence judgement on clone candidates that are similar in structure but actually different in semantics. At the same time, it can analyze and make judgement on implicit abstract data types in the source code which expands the scope of similarity determination. But the problem is that the core issue of method based on PDG — problem of subgraph with same structure is a very difficult algorithmic problem. The current level of research is not able to find a method acceptable in time complexity, so this method is not applicable for large-scale software system. Komondor [9], Krinke [13] and Liu [12]’s methods are representatives of this field. Table 3.4 makes a summary of these three methods.
3.2 Drawbacks of mainstream clone detection methods

Most of the current clone detection methods only focus on detecting similarity of software source code itself during clone detection. These methods pay little attention to relationship between code and this kind of relationship is often called as context information. Methods based on program dependency graph make use of relationship between source codes, but complexity of the detection algorithm is too high to be effectively applied in large-scale software systems. By introducing context information of program, it is able to detect some Type III code clones to some degree. I try to introduce context information of code into clone detection process to improve clone detection methods so that some Type III code clones can be detected, which is based on a existing clone detection tool based on token - \textit{CCFinder}.

3.3 \textit{CCFinder} - clone detection tool based on token

Kamiya [8]'s \textit{CCFinder} is a clone detection tool based on token. It is able to find Type I and Type II code clones with $O(n)$ time complexity ($n$ is the length of source code).

Main steps of clone detection by \textit{CCFinder}:

1. Lexical analysis phase
Each line of source code is transformed into several tokens after analysis by a specific lexer. Tokens of all source code files are concatenated into a token sequence.

2. Normalization phase
Token sequence, which is the result of lexical analysis phase, is normalized according to the programming language in this phase in order to reduce impact by unimportant factors of programming language. For example, generic types of C++ are normalized to normal types.

3. Parameter replacement phase
In this phase, all the variables, types and constants are replaced by a special token. Parameter replacement makes code fragments with different variables, different variable types and different constants able to be detected by detection program. Table 3.5 is an example of parameter replacement for a fraction of C++ code.

<table>
<thead>
<tr>
<th>Original Source Code</th>
<th>Code transformed based on language-dependent rules</th>
<th>Code after parameter replacement</th>
</tr>
</thead>
</table>
| void print_lines (const set &s){
    int c=0;
    set<string>::const_iterator i =s.begin();
    for (;i!=s.end();++i){
        cout<<c<<"","<<"i<<endl;
        ++c;
    }
} void print_value (const vector &v){
    int c=0;
    vector<int>::const_iterator i =v.begin();
    for (;i!=v.end();++i){
        cout<<c<<"","<<"i<<endl;
        ++c;
    }
}                       | void print_lines(const set &s){
                           | int c=0;
                           | const_iterator i =s.begin();
                           | for (;i!=s.end();++i){
                           |    cout<<c<<"","<<"i<<endl;
                           |    ++c;
                           | }
                           | void print_value(const vector &v){
                           | int c=0;
                           | const_iterator i =v.begin();
                           | for (;i!=v.end();++i){
                           |    cout<<c<<"","<<"i<<endl;
                           |    ++c;
                           | }
                           | $p$ $p$ ($p$ $p$&$p$){
                           | $p$ $p$ =$p$;
                           | $p$ $p$ = $p$. $p$.();
                           | for (;$p$!= $p$. $p$.();++$p$){
                           | $p$<<$p$<<$p$<<$p$<<$p$;
                           | ++$p$;
                           | }
                           | $p$ $p$ ($p$ $p$&$p$){
                           | $p$ $p$ =$p$;
                           | $p$ $p$ = $p$. $p$.();
                           | for (;$p$!= $p$. $p$.();++$p$){
                           | $p$<<$p$<<$p$<<$p$<<$p$;
                           | ++$p$;
                           | }
|                       | }                                                  | }                                               |

4. Clone detection phase
The achievement of parameter replacement phase is a single token sequence generated by special symbols replacement. CCFinder adopts a suffix-tree based on substring matching algorithm to find out same substring pairs, which turns out to be the exact clone pairs after being mapped to source code.
Chapter 4 Clone detection method based on context information

4.1 The approach to improve clone detection

Most of current clone detection methods use simple examples to verify its correctness and effectiveness, which are not built on basis of practice. Therefore many researchers point out that some quantitative standards should be used to evaluate clone detection methods [36,37]. Recall, Precision, Complexity are some of the main points. These standards are followed to evaluate the improved clone detection method as well.

4.1.1 Definition of evaluation value

Recall
Recall refers to percentage of the number of detected software clones in total number of software clones in system. This value is used to measure coverage of different types of software clones of the detection method.

Precision
Precision refers to percentage of valid clone pairs in all detected clone pairs. This value is used to measure possibility of detecting invalid clones by the detection method. High precision is important for a good detection method.

Complexity
Complexity refers to complexity of matching algorithm and overall complexity of the detection method. Complexity is an important evaluation value for effectiveness of detection method.

4.1.2 Improvement target

I introduced a clone detection method based on context information, whose target is to improve method based on CCFinder so that it can detect most of Type III clones, which are composed of Type II code clones with a relatively low complexity and a higher recall and precision.

4.2 Introduction of context information

4.2.1 Meaning of introduction of context information

Context information, as the name suggests, is the information of code’s surroundings or relationship between code and its adjacent code. Code clone detection has a significant importance to refactoring and maintenance of large software systems. Without taking context information of code into account, current mainstream code clone detection techniques can detect most of the code clones, but there are a lot of code clones that are pointless for software refactoring. Manually filtering is needed to filter valid code clones for refactoring, which is inefficient and requires a lot of manpower. Introduction of context information can solve this problem to some degree.

4.2.2 Types of context information
Context information discussed in this thesis is mainly categorized into three types: physical distance between two code fragments, positional relationship of code fragments and affinity between two code fragments. Details are discussed in section 4.3.

4.2.3 Acquisition of context information

The main way of acquiring context information of program is to scan and analyze nearby code fragments. Different type of context information is acquired in different ways. Details are discussed in section 4.4.

4.3 Definition of context information

4.3.1 Physical distance

A code fragment can be uniquely identified by following three values: name of source file (assuming source file's name is unique in original system, otherwise use source file sequence number instead), starting line number of code fragments in source file and end line number of code fragments in source file. Distance between code fragments is defined as following:

- If two code fragments belongs to two different source files, the distance between these two code fragments is infinitely great.
- If two code fragments belongs to one source file, and starting line number of code fragment 1 is less than starting line number of code fragment 2. Then physical distance between code fragment 1 and code fragment 2 is 0 if end line number of code fragment 1 is equal or greater than starting line number of code fragment 2. Physical distance is the difference between end line number of code fragment 1 and starting line number of code fragment 2 if end line number of code fragment 1 is less than starting line number of code fragment 2.

Table 4.1 Example of physical distance between two code fragments

<table>
<thead>
<tr>
<th>Code Fragment 1 (hello.cpp)</th>
<th>Code Fragment 2 (hello.cpp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>line 90: if (a!=b)</td>
<td>line 105: if (i!=j)</td>
</tr>
<tr>
<td>line 91: {</td>
<td>line 106: {</td>
</tr>
<tr>
<td>line 92: a=a+b;</td>
<td>line 107: i=i+j;</td>
</tr>
<tr>
<td>line 93: c=a;</td>
<td>line 108: m=m;</td>
</tr>
<tr>
<td>line 94: }</td>
<td>line 109: }</td>
</tr>
</tbody>
</table>

Table 4.1 illustrates two code fragments in one source file. According to definition of physical distance, the distance between these two code fragments is 11.

4.3.2 Positional relationship between code fragments

It's an important task for software refactoring to refactor existing duplicate code to functions that are reusable. But it is quite difficult to refactor two code fragments that do not belong to the same function and are not code clone pairs either. I introduce positional relationship into the improved clone detection method. The following is the definition:

- If two code fragments belong to one function of the same source file, than the positional relationship value is 1.
- If two code fragments do not belong to the same source file or belong to different functions of the same source file, then the positional relationship value is 0.
Table 4.2 gives an example of two code fragments in the same source file. According to definition of positional relationship between code fragments, the positional relationship value of these two code fragments is 1.

<table>
<thead>
<tr>
<th>Code Fragment 1 (hello.cpp)</th>
<th>Code Fragment 2 (hello.cpp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>line 75: void func(int a)</td>
<td>line 75: void func(int a)</td>
</tr>
<tr>
<td>line 76: {</td>
<td>line 76: {</td>
</tr>
<tr>
<td>.....</td>
<td>.....</td>
</tr>
<tr>
<td>line 90: if (a!=b)</td>
<td>line 135: if (x==y)</td>
</tr>
<tr>
<td>line 91: {</td>
<td>line 136: {</td>
</tr>
<tr>
<td>line 92: a=a+b;</td>
<td>line 137: x++;</td>
</tr>
<tr>
<td>line 93: c=a;</td>
<td>line 138: }</td>
</tr>
<tr>
<td>line 94: }</td>
<td>line 139: }</td>
</tr>
<tr>
<td>.....</td>
<td>.....</td>
</tr>
<tr>
<td>line 200: }</td>
<td>line 200: }</td>
</tr>
</tbody>
</table>

### 4.3.3 Affinity between code fragments

The more common variables are shared between code fragments, the relationship of these two code fragments is closer, which makes more meaningful code fragments for software refactoring. Introducing affinity between code fragments is beneficial to detecting code clones with more semantics and more useful for refactoring. The following is definition of affinity between code fragments:

- The affinity between two code fragments is the number of common variables referenced in two code fragments.

Table 4.3 lists two code fragments. According to definition of closeness, the affinity of these two code fragments is 3 (common variables are a, b, m).

<table>
<thead>
<tr>
<th>Code Fragment 1 (hello.cpp)</th>
<th>Code Fragment 2 (hello.cpp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>line 90: if (a!=b)</td>
<td>line 105: if (a&gt;3)</td>
</tr>
<tr>
<td>line 91: {</td>
<td>line 106: {</td>
</tr>
<tr>
<td>line 92: a=a+b;</td>
<td>line 107: a++;</td>
</tr>
<tr>
<td>line 93: c=a;</td>
<td>line 108: b--;</td>
</tr>
<tr>
<td>line 94: d=m-n;</td>
<td>line 109: e=b;</td>
</tr>
<tr>
<td>line 95: }</td>
<td>line 110: m=a;</td>
</tr>
<tr>
<td></td>
<td>line 111: }</td>
</tr>
</tbody>
</table>

### 4.4 Collection of context information

We can get a collection S composed of Type II code clone pairs after detecting clones using CCFinder. It is the collection S that we use to collect context information.
4.4.1 Collection of physical distance

Clone detection program saves line information and file information related with tokens during the phase of lexical analysis, so physical distance is easy to be calculated according to the definition.

4.4.2 Collection of positional relationship between code fragments

Collection of positional relationship between code fragments is different from collection of physical distance. With the knowledge of source file information of source code, starting line number and termination line number, program traces back in the source code to find function name to which the code fragment belongs. It is with no difficulty to get positional relationship between code fragments after the function names are retrieved.

4.4.3 Collection of affinity between code fragments

Collection of affinity is simple and straightforward. Firstly, different variables of two code fragments are saved into two variable sets respectively, then affinity is the number of common element of two variable sets.

4.5 Detection algorithm

4.5.1 Overview

A software clone detection tool - CCFinder is introduced, which is able to successfully detect Type II code clones that can be applied in the process of software refactoring. The detection algorithm discussed in this thesis detects Type III clones, which is a result of combination of Type II clones. The algorithm uses context information generated from the detection result of Type II clones by CCFinder. There are mainly three reasons to adopt this algorithm to detect Type III code clone:

1. Technique of detecting Type II code clone by CCFinder has been proved that it is able to be applied in large-scale system.
2. This algorithm is able to detect some Type III code clones coming from Type II code clones which are of great importance to software refactoring.
3. There are already mature approaches to do refactoring for Type III code clones, so using Type III code clone in refactoring is feasible.

4.5.2 Pseudocode of detection algorithm

This section describes a pseudocode of the detection algorithm in informal high-level words.

Begin:

1) Input: original source code
do Type II clone pairs detection with CCFinder
Output: a collection of Type II clone pairs (B1, B2), (B3, B4) …. (B2n-1, B2n)

2) Input: result of step 1
   sort Type II clone pairs based on line numbers
   Output: a collection of sorted Type II clone pairs

3) Input: result of step 2
   Begin: for each sorted Type II clone pair
   look for candidate Type II clone pairs based on context information
   End
Output: several groups of candidate Type II clone pairs

4) Input: result of step 3
   Begin: for each group of candidate Type II clone pairs
         combine Type II clone pairs into a Type III clone pair
   End
   Output: a collection of Type III clone pairs

End

4.5.3 Details of detection algorithm

The main theory of clone detection algorithm adopted by this thesis is using the detection result of CCFinder, which is a collection of Type II code clone pairs, to filter some candidate Type II code clone pairs. The theory suggests using context information of clone pairs and combining the remaining Type II code clone pairs based on some criteria to generate the final collection of Type III code clone pairs. The following is main steps of the algorithm:

1. Generating collection of Type II code clone pairs. In this phase, CCFinder is applied to do clone detection on source codes and generates a collection of Type II code clone pairs (referenced as S), which are similar to pairs like (B1, B2). B1 represents code fragment 1 in a clone pair and B2 is code fragment 2 in the same clone pair. Each code fragment can be uniquely identified by the following three values - FileNumber, StartLineNumber, EndLineNumber.

2. Sorting. Type II clone pairs in collection S is sorted in this phase. The following is the sorting rule:
   1. Clone pairs are sorted by B1’s FileNumber from the smallest to the largest.
   2. If B1’s FileNumber of two clone pairs is the same, then two clone pairs are sorted by B1’s StartLineNumber from the smallest to the largest.
   3. If B1’s FileNumber and B1’s StartLineNumber of two clone pairs are both the same, then two clone pairs are sorted by B2’s FileNumber from the smallest to the largest.
   4. If B1’s FileNumber and B1’s StartLineNumber of two clone pairs are both the same, and B2’s FileNumber of two clone pairs is also the same, then two clone pairs are sorted by B2’s StartLineNumber from the smallest to the largest.

3. Looking for candidate Type II code clone pairs based on context information. In this phase, Type II code clone pairs in the sorted collection are filtered to meet specific requirements that are related with context information. For example, if (B1, B2) and (B3, B4) are two clone pairs of phase 2 and context information of these two clone pairs meeting the following requirements, then these two clone pairs can be treated as candidate Type II code clone pairs. The following is an example of requirements:
   1. Physical distance between B1 and B3 and physical distance between B2 and B4 is both less than 10
   2. Positional relationship value between B1 and B3 and positional relationship value between B2 and B4 is both 1.
   3. Affinity between B1 and B3 and affinity between B2 and B4 is both not less than 10
The requirements on context information for candidate Type II code clone pairs may be different for different software systems. The requirements have a great impact on the number and quality of Type III code clones that are finally detected

4. Combing candidate Type II code clone pairs. Assuming that (B1, B2) and (B3, B4) are candidate Type II code clone pairs found in phase 3, a new Type III code clone pair will be generated if these two Type II code clone pairs are combined. The following is the rule for combination:
1. Assuming \( B_1 \) is sorted before \( B_3 \) according to the sorting rule defined in phase 2 and \( B_1' \) is new code fragment combined by \( B_1 \) and \( B_3 \), then \( B_1' \) has the same FileNumber as \( B_1 \) and \( B_3 \), the same \textit{StartLineNumber} as \( B_1 \) and the same \textit{EndLineNumber} as \( B_3 \). Otherwise, \( B_1' \) has the same \textit{StartLineNumber} as \( B_3 \) and the same \textit{EndLineNumber} as \( B_1 \).

2. Assuming \( B_2 \) is sorted before \( B_4 \) according to the sorting rule defined in phase 2 and \( B_2' \) is new code fragment combined by \( B_2 \) and \( B_4 \), then \( B_2' \) has the same FileNumber as \( B_2 \) and \( B_4 \), the same \textit{StartLineNumber} as \( B_2 \) and the same \textit{EndLineNumber} as \( B_4 \). Otherwise, \( B_2' \) has the same \textit{StartLineNumber} as \( B_4 \) and the same \textit{EndLineNumber} as \( B_2 \).

New code clone pair \((B_1', B_2')\) is the Type III code clone pair detected by the detection algorithm.
Chapter 5 Implementation of clone detection method based on context information

I combined the knowledge of chapter 3 and chapter 4 in this chapter and implemented the improved code clone detection algorithm by using Java and designs some experiments to analyze the improved clone detection algorithm.

5.1 Development environment

Implementation of improved clone detection algorithm is written in Java programming language. Java programming language is a concurrent, class-based, object-oriented and type-safe computer programming language which is platform-independent and efficient. What's more, there are many java libraries and frameworks. Therefore, Java is chosen as the development language for the implementation.

5.2 System architecture

According to the function and design of the detection tool, the tool is divided into three modules: module for extraction of Type II code clones, module for extraction of context information and module for detection of Type III code clones.

1. Module for extraction of Type II code clones is responsible for extracting Type II code clones by using CCFinder, which is used in the other two modules.
2. Module for extraction of context information is responsible for extracting context information of code clones to be used to detect Type III clones.
3. Module for detection of Type III code clones is responsible for making use of results generated by the other two modules to generate new Type III code clones.

In accordance with the difference between the different modules, three java packages are designed: com.clone.ccfinder, com.clone.context, com.clone.detect. Among these three packages, com.clone.ccfinder is the package containing Java classes that implement functions of extracting Type II code clone pairs by using CCFinder and offer unified interfaces for the other two modules. com.clone.context is the package responsible for extracting context information between code fragments and package com.clone.detect combines information provided by package com.clone.ccfinder and package com.clone.context to detect new Type III code clones. The overall design is shown in Figure 5.1:

26
5.2.1 Module for extraction of Type II code clones

Module for extraction of Type II code clones is encapsulated in the package of `com.clone.ccfinder` which is mainly responsible for detecting Type II code clones using `CCFinder`.

`CCFinder` is an open-source code clone detection tool and its token-based detection algorithm is also open-source which is of a high efficiency. After reading through source code of `CCFinder`, I found that its algorithm is implemented in C++ language though graphic user interface is implemented in Java. Because Java is chosen to implement the tool, JNI technique in Java should be used to get the detection result of `CCFinder`. Figure 5.2 is the design diagram of main classes of module for extraction of Type II code clones.

Class `CodeBlock` represents code fragments and class `ClonePair` refers to a clone pair. Class `CCFinderUtility` makes function calls to `CCFinder` tool with a series of parameters and returns code clone pairs detected by `CCFinder`. 
5.2.2 Module for extraction of context information

Module for extraction of context information is encapsulated in the package `com.clone.context`, which is mainly responsible for extraction of three types of context information defined in chapter 4. Figure 5.3 is the design diagram of main classes of module for extraction of context information.

Class `CodeBlock` is imported from package `com.clone.ccfinder` and represents code fragment. Class `ContextInfo` encapsulates context information of two code fragments for a good extensibility — it is easy to extend function of the program by modifying class `ContextInfo` if other types of context information is found that can be introduced into clone detection. Class `ContextInfoCollector` accepts two `CodeBlock` type parameters and calls class `DistanceInfoCollector` to calculate physical distance of two code fragments and calls class `AffinityInfoCollector` to calculate affinity between two code fragments so that context information of two code fragments can be retrieved and used by module of clone detection.

Figure 5.3 Design diagram for main classes of module for extraction of context information
5.2.3 Module for detection of Type III code clones

Module for detection of Type III code clones is encapsulated in the package `com.clone.detect` and is mainly responsible for combining Type II code clone pairs and context information, which are results of the other two modules, to generate new Type III code clones. Figure 5.4 is the design diagram of main classes of module for detection of Type III code clones.

![Design diagram for main classes of module for detection of Type III code clones](image)

Class `CandidateClonePairs` represents candidate Type II code clone pairs filtered by specific requirements on context information. Class `ClonePairFilter` implements interface `ContextInfoFilter`, which contains three function interfaces of calculating physical distance, positional relationship and affinity. Class `ClonePairFilter` contains a static method `filter` which calls methods of class `ContextInfoCollector` of the package `com.clone.context` to get context information and uses the context information to filter some clone pairs and finally produces objects of class `CandidateClonePairs`. Class `ClonePairCombinator` makes use of result of class `ClonePairFilter` to combine candidate clone pairs and finally generates new Type III code clone pairs.

5.3 Detection flow
The first step of detection flow is reading source code files to be detected. Type II code clone detection will be proceeded after reading source files. User needs to specify type of programming language of the source files and also can modify some detection parameters, such as minimum clone length (by default is 25). After result of Type II code clone detection is retrieved, extracting context information will be proceeded. After extraction of context information is done, the program will filter these Type II code clone pairs based on threshold value related with context information, which is also set by user. Finally Type III code clone pairs are generated from the filtering results.

5.4 Analysis of experiment result
To verify effectiveness of the improved method, I choose several mainstream software as target systems of detection, which are Jhotdraw, Linux kernel and PostgreSQL, compares and analyzes the results with experiment result of CCFinder.

5.4.1 Experiment environment and target systems
Experiment in this thesis is performed on a laptop whose memory capacity is 4G and CPU is Intel Core™ i5-480M 2.67GHz. The running time of an experiment is measured by the timing tool offered by operating system and is the average value of several experiments.

Jhotdraw, Linux kernel and PostgreSQL are popular test target systems in the field of code clone detection research. Table 5.1 shows the basic information of these three systems:
Table 5.1 Basic information of target system

<table>
<thead>
<tr>
<th>System Name</th>
<th>Number of Lines</th>
<th>Number of Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jhotdraw</td>
<td>50K</td>
<td>1700</td>
</tr>
<tr>
<td>Linux kernel</td>
<td>3M</td>
<td>3964</td>
</tr>
<tr>
<td>postgreSQL</td>
<td>3M</td>
<td>4352</td>
</tr>
</tbody>
</table>

If \((B1,B2)\) and \((B3,B4)\) are two candidate Type II code clone pairs, then \(B1, B2, B3, B4\) all represent a code fragment. The filtering requirements of context information applied in the experiments are listed below:

1. Physical distance between \(B1\) and \(B3\) and physical distance between \(B2\) and \(B4\) both are less than 10.
2. Positional relationship value between \(B1\) and \(B3\) and positional relationship value between \(B2\) and \(B4\) are both 1.
3. Affinity between \(B1\) and \(B3\) and affinity between \(B2\) and \(B4\) are both greater than 5.

5.4.2 Calculating evaluation value

**Precision**

Firstly, randomly select 20 groups of clone pairs from the detection results of each system of two detection methods and then perform a manual assessment to get number of correct clone pairs.

Estimated precision value is the result of number of correct clone pairs divided by total clone pairs that are detected finally.

**Recall**

After estimated precision value is calculated, we can get the number of valid code clone pairs by multiplying precision value by the number of code clone pairs detected by the method.

Recall value is the result of number of valid code clone pairs divided by total valid code clone pairs (provided by other detection tools) in the system.

5.4.3 Experiment result and analysis

Experiment result is shown with tables and histograms below. It is noteworthy that code clones in the report includes Type II code clone and Type III code clone.
Table 5.2 Detection result of CCFinder of target systems

<table>
<thead>
<tr>
<th>Target System</th>
<th>Detection time (in seconds)</th>
<th>Number of detected code clones</th>
<th>Precision</th>
<th>Recall</th>
<th>Estimated total number of code clones in system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jhotdraw</td>
<td>35</td>
<td>50</td>
<td>70%</td>
<td>44%</td>
<td>80</td>
</tr>
<tr>
<td>Linux kernel</td>
<td>105</td>
<td>5125</td>
<td>71%</td>
<td>45%</td>
<td>8000</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>104</td>
<td>4000</td>
<td>68%</td>
<td>30%</td>
<td>9000</td>
</tr>
</tbody>
</table>

Table 5.3 Detection result of improved method of target systems

<table>
<thead>
<tr>
<th>Target System</th>
<th>Detection time (in seconds)</th>
<th>Number of detected code clones</th>
<th>Precision</th>
<th>Recall</th>
<th>Estimated total number of code clones in system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jhotdraw</td>
<td>42</td>
<td>59</td>
<td>68%</td>
<td>50%</td>
<td>80</td>
</tr>
<tr>
<td>Linux kernel</td>
<td>125</td>
<td>5734</td>
<td>67%</td>
<td>48%</td>
<td>8000</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>116</td>
<td>4463</td>
<td>63%</td>
<td>31%</td>
<td>9000</td>
</tr>
</tbody>
</table>

Figure 5.6, 5.7, 5.8 shows detection results of CCFinder and the improved method from the point view of running time, precision and recall.

It can be inferred from figure 5.6 that running time of the improved method is more than running time of CCFinder, which is caused by extra calculation work after introducing context information in the process of clone detection and it is inevitable. The rate of increase in running time is related with size of the software system. Comparing with the other two systems, Jhotdraw, which has a smaller size, has a smaller rate of increase in running time.
Figure 5.6 Comparison of running time of two detection methods

Figure 5.7 Comparison of precision of two detection methods
Result in Figure 5.7 shows that precision of the improved detection method decreases slightly. The main reason is that there are many false-positives in the Type III code clone of the detection result, which is maybe a result of imperfect filtering rule of context information.

Result in Figure 5.8 shows that the improved method has an increase in recall. The main reason is that more Type III code clones are found by the improved detection algorithm, which increases the total amount of detected code clones.

5.4.4 Conclusions

We can make the following conclusions from previous experiments and their results.

**Increase in time complexity**
After context information is introduced, program has an increase in running time because of extra calculation work. But the rate of increase is not large comparing to total running time. Increase of running time to some extend is acceptable under the premise of increasing recall value of clone detection method.

**Decrease in precision**
Detection precision of the improved method decreases slightly. The reason of the situation is that more false-positive clone pairs are detected by the improved detection method, which may be a result of inaccurate context information or inappropriate filtering rule of context information. The exact reason needs to be found out in the future researches.

**Increase in recall**
The improved method is able to detect Type III code clones combined from Type II code clones. Though precision of detection decreases slightly, but more code clones are detected, which is of positive significance.

The improved detection algorithm and its implementation is proved to be valuable through verification and analyze of the above experiments.
Chapter 6 Conclusion and outlook

6.1 Conclusion

I made a summary first of current mainstream software clone detection techniques, then discussed disadvantages of existing software clone detection methods from the aspect of theory and practice. Then I concluded that disadvantages are mainly caused by lack of context information of program in the process of code clone detection. Finally I introduced context information into the process of clone detection based on CCFinder, which is helpful for detection of Type III code clones.

In the aspect of theory, I discussed the necessity of introducing context information into clone detection through analyzing existing clone detection techniques and came up with reasonable method to represent context information.

In the aspect of practice, I developed a new clone detection tool which introduces context information into clone detection on the basis of clone detection tool CCFinder, designed and performed some experiments to prove introduction of context information is really helpful for detection of Type III code clones. Though Recall of clone detection increases, introduction of context information causes decrease in Precision of clone detection. A balanced point should be found for Recall and Precision.

6.2 Future work

Though research of this thesis has made an initial success, a lot of future work still needs to be done. Some important parts are discussed in the section below:

As discussed in previous section, by introducing context information into clone detection, it is able to detect some Type III code clones and it is quite obvious that Recall of clone detection increases. However, Precision of clone detection decreases at the same time. It is a big challenge for our future work.

On the other hand, the improved detection method is not able to detect Type IV code clone, which is also called semantic clone. Theoretically speaking, context information is helpful for detection of this type of clone. Context information is successfully introduced and applied into the process of clone detection in this thesis. The detection of Type IV code clones may be not a tough task if we can find a way to make use of context information, which should be researched in the future.
Chapter 7 Bibliography


[12] Chao Liu, Chen Chen, Jiawei Han and Philip S. Yu. GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. In the Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD’06), pp. 872-881, Philadelphia, USA, August 2006.


[16] Lerina Aversano, Luigi Cerulo, and Massimiliano Di Penta. How Clones are Main-tained: An
Empirical Study. In Proceedings of the 11th European Conference on Soft-ware Maintenance and
Reengineering (CSMR'07), pp. 81-90, Amsterdam, the Netherlands, March 2007.


code by frequent itemset techniques. In Proceedings of the 4th IEEE International Workshop
Source Code Analysis and Manipulation (SCAM'04), pp. 128135, Chicago, IL, USA, September
2004.

[19] Zhenmin Li, Shan Lu, Suvda Myagmar, Yuanyuan Zhou. CP-Miner: A Tool for Finding Copy-
Operating System Design and Implementation (OSDI’04), pp.289-302, SanFrancisco, CA, USA,
December 2004.

Advanced Clone-analysis to Support Object-oriented System Refactoring. In Proceedings of the
7thWorking Conference on Reverse Engineering (WCRE’00), pp. 98-107, Brisbane, Qld., Australia,
November 2000.

Working Conference on Reverse Engineering (WCRE’99), pp. 326-336, Atlanta, GA, USA, October
1999.

Measuring Clone Based Reengineering Opportunities. In Proceedings of the 6th International
Software Metrics Symposium (METRICS’99), pp. 292-303, Boca Raton, Florida, USA, November
1999.

IEEE International Conference on Program Comprehension (ICPC’06), pp. 56-68, Athens, Greece,
June 2006.


Based Clone Detection with Flexible Tokenization. In Proceedings of the Joint Meeting of the
European Software Engineering Conference and Symposium on the Foundations of Software

Proceedings of the 10th European software engineering conference held jointly with 13th ACM
SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE’ 05), pp.

[27] Hamid Basit, Damith Rajapakse, Stan Jarzabek. Beyond Templates: a Studyof Clones in the
STL and Some General Implications.In Proceedings of the 27th Interna-tional Conference on
Software Engineering (ICSE’05), pp. 15-21, St. Louis, Missouri, USA, May 2005.


