Phoenix-Based Clone Detection Using Suffix Trees

Robert Tairas and Jeff Gray
Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, AL 35294 USA
{tairasr, gray}@cis.uab.edu

ABSTRACT

A code clone represents a sequence of statements that are duplicated in multiple locations of a program. Clones often arise in source code as a result of multiple cut/paste operations on the source, or due to the emergence of crosscutting concerns. Programs containing code clones can manifest problems during the maintenance phase. When a fault is found or an update is needed on the original copy of a code section, all similar clones must also be found so that they can be fixed or updated accordingly. The ability to detect clones becomes a necessity when performing maintenance tasks. However, if done manually, clone detection can be a slow and tedious activity that is also error prone. A tool that can automatically detect clones offers a significant advantage during software evolution. With such an automated detection tool, clones can be found and updated in less time. Moreover, restructuring or refactoring of these clones can yield better performance and modularity in the program.

This paper describes an investigation into an automatic clone detection technique developed as a plug-in for Microsoft's new Phoenix framework. Our investigation finds function-level clones in a program using abstract syntax trees (ASTs) and suffix trees. An AST provides the structural representation of the code after the lexical analysis process. The AST nodes are used to generate a suffix tree, which allows analysis on the nodes to be performed rapidly. We use the same methods that have been successfully applied to find duplicate sections in biological sequences to search for matches on the suffix tree that is generated, which in turn reveal matches in the code.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *restructuring, reverse engineering, and reengineering*

General Terms

Algorithms, Management, Performance, Design, Reliability, Experimentation, Languages, Theory.

Keywords

Clone Detection, Code Clones, Suffix Trees, Software Analysis

1. INTRODUCTION

When a section of code is duplicated in more than one location in a program, that section of code and all of its duplicates are considered code *clones*. Clones are typically generated because a programmer copies a section of code and pastes it into another part of the same program. This may be done because the copied section of code performs some functionality correctly. Rather than rewriting the code from scratch, it is much simpler to copy this code and use it in another part of the program. This practice often results in maintenance problems at later stages of development.

Another reason for the existence of code clones relates to the aspect-oriented programming [9] notion of a dominant decomposition, where one functionality (or concern) dominates another [12]. The two concerns crosscut each other and the code for the "weak" functionality must be scattered throughout the program. This produces clones of the same functionality in various parts of the program.

Research has shown that a considerable percentage of code in large-scale computer programs are clones [2]. During the maintenance stage it would be beneficial to determine if code clones occur in a program. If a section of code is known to be cloned, then when that section needs to be updated, its clones will have been identified and can be updated as well. With the knowledge of the clones, restructuring or refactoring of these clones could be done to enhance the quality of the program. In terms of aspect-oriented programming, the code that is dominated by other concerns could be extracted and made into an aspect, which may enhance the maintainability of the program [3].

Code clones can be discovered manually by scavenging through the program source and identifying duplicates one by one. Depending on the size of the program, this manual process can become tedious and labor intensive. An automatic clone detection tool can be beneficial by reducing the time and effort needed to find clones. Various studies have undertaken the development of clone detection tools by examining different levels of program representation. These studies have used text-based, token-based, AST-based, program dependence graph-based, metrics-based, and information retrieval-based representations [3].

The clone detection tool described in this paper utilizes an abstract syntax tree (AST) representation of the program. An advantage of evaluating the AST representation of a program is that the AST generalizes the parse tree by simplifying the structure of the tree, without losing the overall definition of the program. This reduces the amount of data that will need to be evaluated to find code clones.

The individual nodes of the AST are of particular interest. The nodes are extracted from the AST and a suffix tree is generated from these nodes. Suffix trees have been successfully used in biological sequence matching [4][8]. The method of using suffix trees to find duplicate biological sequences is applied to the search for duplicate node sequences of the AST. Exact matching code clones can be detected using this method.

In the clone detection tool comparison experiment at the *First International Workshop on Detection of Software Clones*[1], clones were separated into three categories:

- Exact copies, with no differences between them
- Parameterized copies, where variable and function calls can have different names and/or types have changed
- Modified copies, where some modification is done, such as adding or deleting lines of code

The clone detection tool described in this paper focuses on the first two categories of clones with a slight modification of the second category. For the second category, variables and function calls in exact matching functions can be named differently, but the types of these must be the same. Future work related to the last category of clones is discussed later.

Our tool is developed as a plug-in for Microsoft's Phoenix framework [11], which supports the development of compilers and software analysis tools. It is the basis of all future Microsoft compiler technologies. Although initially offered to academia to aid in their research, the Phoenix framework is intended to be an industrial-strength framework for production-level development.

The following section discusses an algorithm for finding exact matching function-level clones. Section 3 shows how the clone detector is implemented in the Microsoft Phoenix framework. Section 4 reports on two case studies using the clone detection tool. Section 5 compares related work to the approach used in this paper. Section 6 concludes the paper and discusses future work.

2. EXACT MATCHING ALGORITHM

2.1 The Original Suffix Tree

As its name suggests, a suffix tree is a tree of suffixes. A suffix tree of a string is generated from the suffixes of that string. For each suffix of a string, a path is made from the root to a leaf. This is done by evaluating each character in the suffix and generating new edges when no existing edges that represent the character in the suffix tree exists [6]. This is the characteristic of the suffix tree that is useful in string matching, because duplicate patterns in the suffixes will be represented by a single edge in the tree. A string *abcdabe$* is represented by the suffix tree in Figure 1.

The pattern *ab* is represented by a single edge. Two suffixes pass through this edge (i.e., they both start with the substring *ab*). These two suffixes are *abcdabe$* and *abe$*. The split at the end of this edge continues the two suffixes where the next character differs between the two suffixes. The last character, *$*, is a special terminating character that identifies the end of the string. By looking at the suffixes that pass through the edge that represents the pattern *ab*, the location of this string pattern can be determined.

[1] http://www.informatik.uni-stuttgart.de/ifi/ps/clones/

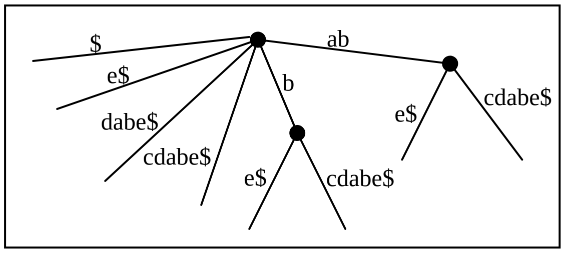

Figure 1. Suffix tree of *abcdabe$*.

The use of suffix trees to search for duplicate patterns is not limited to just one string. Searching for duplicate patterns in multiple strings is also possible. The suffix tree used to search for duplicate patterns in multiple strings is generated from the concatenation of the strings. For example, to generate the suffix tree of two identical strings *abgf* and *abgf*, the two strings are concatenated into one string *abgf$abgf#*, with *$* and *#* as the special characters that determine where each string terminates. This new string is evaluated as a single string and the same process used in the previous example is used on this string to generate the suffix tree. The result is the suffix tree in Figure 2. Duplicate patterns can be identified in this suffix tree and with some additional processing, the individual strings that contain these patterns can be determined.

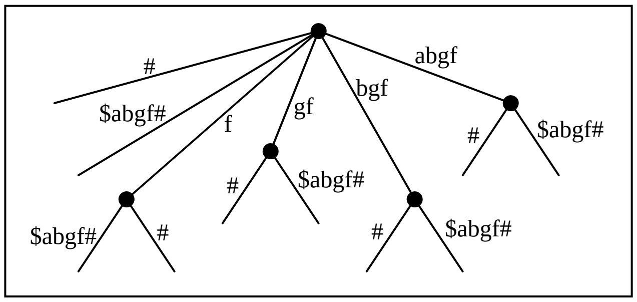

Figure 2. Suffix tree of *abgf$abgf#*.

The edge labeled "abgf" represents two suffixes of the concatenated string that start at the beginning positions of each individual string. That is, the first suffix is the whole string *abgf$abgf#*, which starts at the beginning position of the first individual string. The second suffix is the substring *abgf#*, which starts at the beginning position of the second individual string. This edge is split at the end into two edges because the next characters in the suffixes are different. The differing characters are the terminating characters of the two individual strings, *$* and *#*. The existence of this type of edge determines that the two individual strings are exact duplicates of each other.

2.2 Suffix Tree Alteration

Our approach applies the evaluation of suffix trees (as described in the previous subsection) to search for functions in a program that are exact duplicates of each other. The first step is to replace the string with a representation of the functions in a program. This is where the nodes of an AST are used. Figure 3 displays the AST that represents function #1 in Figure 4.

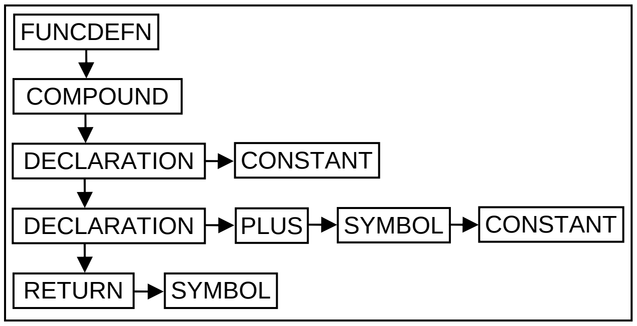

Figure 3. Abstract syntax tree nodes

In Figure 3, the AST is flattened and the nodes are connected together to produce the sequence [FUNCDEFN] [COMPOUND] [DECLARATION] [CONSTANT] [DECLARATION] [PLUS] [SYMBOL] [CONSTANT] [RETURN] [SYMBOL]. This sequence of node names will become the string whose suffixes will be used to generate the suffix tree.

To represent all the functions in a program, the AST node sequences of each function is concatenated together to produce one long sequence. Special terminating nodes are inserted between each function representation in the sequence of nodes. A suffix tree is generated from this sequence and by using the method of searching for certain edges in the suffix tree described earlier, functions that are duplicates of each other can be determined.

Function #1:	`int main() {` `int x = 1;` `int y = x + 5;` `return y;` `}`
Function #2:	`int main() {` `int x = 3;` `int y = x + 5;` `return y;` `}`
Function #3:	`int main() {` `int x = 1;` `int y = y + 5;` `return y;` `}`

Figure 4. Example functions

2.3 Potential False Positives

It is not sufficient to determine if two functions exactly match based only on the suffix trees of AST node names. Several situations can lead to false positives (i.e., clones reported as exact matches, but upon further observation are not).

It is possible that the same sequence of AST node names can represent a function that is not exactly the same. For example, observe function #1 in Figure 4. The AST node sequence, which consists of the node names, will be identical to the sequence for function #2. The two functions would be considered exact duplicates. However, the constant values that are assigned to variable *x* in the two functions are different (i.e., one is set to 1 and the other is set to 3).

Another problem that can arise occurs when the variable locations differ from one function to another even if their node sequences are the same. Function #3 in Figure 4 demonstrates this situation. The statement y = x + 5 in function #1 will have the same node sequence as the statement y = y + 5 in function #3 (i.e., the similar sequence is [DECLARATION] [PLUS] [SYMBOL] [CONSTANT]). However, the lines are not exact matches, because the line in function #3 does not contain an *x* variable.

If the suffix tree method of finding duplicates is used, all three functions would be reported incorrectly as duplicates. In order to account for these situations, an additional step is added after duplicates are reported from the suffix tree. This step will observe the *types* and *values* of the nodes of the AST and the positions of the variables in the function. These additional steps are discussed further in Section 3.

2.4 Sketch of Clone Detection Algorithm

The following algorithm represents the approach used in our clone detection tool.

```
/* Generate node sequence */
For each node in the AST:
        Add the node to the sequence of nodes
        If node is the end of a function
        definition:
                Add a terminating node to the
                sequence of nodes

/* Generate suffix tree */
For each suffix of the sequence of nodes:
        Generate a path from the root to a leaf
        combining the path with existing edges
        when the edges represent the same sequence
        of nodes

/* Look for duplicate functions */
For each leaf that represents a suffix that starts
with a function definition node:
        Traverse the path up to the root
        If the last edge before the root
        represents all the nodes of the
        function and more than one terminating
        nodes are found where the edge splits:
                Group the functions associated with
                these terminating nodes together

/* Additional check on duplicate groups */
For each group of duplicate functions:
        Check whether constant values and variable
        positioning match each other
```

3. IMPLEMENTATION DETAILS

3.1 Microsoft Phoenix

The clone detection tool described in this paper is implemented as a plug-in for the Microsoft Phoenix framework. Although this framework has been offered to academia to aid in the research of compilers and software analysis tools, it is also targeted for developers of production-level compilers and tools. Phoenix is a joint project between the Visual C++, Microsoft Research, and .NET Common Language Runtime groups at Microsoft and is poised to be the basis for the next generation of Microsoft compilers [11].

Phoenix is primarily a framework for the backend of a compiler, where optimization and code generation tasks are performed in a customized manner. In Phoenix, the compiler tasks are divided into phases that are executed sequentially according to a specified list. By separating tasks into phases, Phoenix allows customization in any part of the sequence. Custom analysis tools can be developed and included as a specific phase in the process. A new phase is inserted into Phoenix by way of a library (DLL) module that is a plug-in for the compiler. Our clone detection tool is written as a custom phase that is plugged into Phoenix. Figure 5 is a graphical representation of the process.

In Figure 5, an example program called example.c is consumed by the C/C++ Frontend, which is included in Phoenix. This produces example.ast, which contains the AST of the source code in example.c. This AST file is then consumed by the Phoenix Backend. In addition to the AST file, a plug-in called clones.dll is included. This plug-in represents our clone detector. The output from the Phoenix Backend is a report of clones found in example.c.

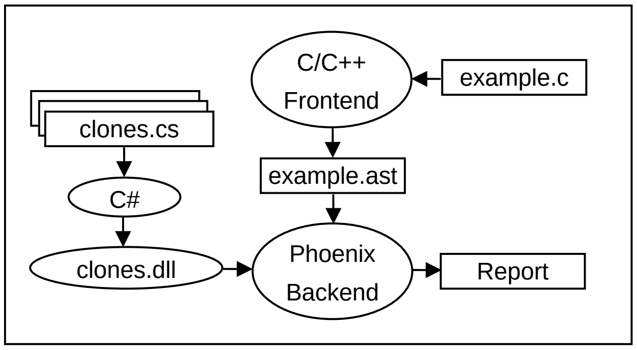

Figure 5. Clone detection plug-in for Phoenix

3.2 Detecting Code Clones

A major part of the clone detection tool is the implementation of a mechanism to generate and evaluate suffix trees. The method used to generate the suffix tree follows a naïve approach. This approach takes $O(m^2)$ time to build a suffix tree, where m is the length of the string or sequence. Linear-time construction of suffix trees can be done using either Ukkonen's or Weiner's method [6]. Building suffix trees with these methods requires more complex procedures. However, they could be implemented if it is desired to reduce the processing time of suffix tree construction.

The program representation in the form of an AST is obtained from an object provided by the Phoenix compiler. The AST consists of *Node* objects that contain information about the node such as name, type, and value. The names of the nodes are used to generate the suffix tree. The type and value of the node is used in the second step of the process when the clone detector examines groups of reported clones to find exact matching clones.

While traversing the AST, any encounter with a "function definition" node is noted. When the end of the function definition is reached in the sequence of nodes, a special terminating node is inserted into the sequence. This terminating node is a custom node that is not part of the node collection provided by Phoenix.

A suffix tree is generated from the sequence of nodes (including terminating nodes). Because each suffix of the sequence is evaluated, there will be a path from the root to a leaf for each suffix. The suffixes whose first nodes are the starting nodes of the functions are of particular interest. The paths that represent these suffixes are traversed from the leaves to the root. Paths that converge (at the top of the tree) into a single edge that represents all the nodes of one or more functions are considered duplicates. These clones are stored together to be evaluated further.

The next step of the process constructs a separate suffix tree of the nodes for each group of duplicate functions reported in the first step. The difference in the construction of the suffix tree compared to the first construction is that not only are the node names compared, but also their type, value, and variable position. Nodes are considered identical if their types are the same, their values are the same, and the variable represented by the node is at the same position in the functions. These checks are applied on the nodes where applicable. The result of these two steps is a list of functions that are exact duplicates of each other.

4. CASE STUDY EXAMPLE

Our clone detection implementation was experimentally applied on two C programs varying in size. The first program was Abyss[2], which is a small web server written in approximately 1500 LOC. The second program was Weltab[3], which is an election results program written in approximately 11K LOC. Weltab was used as part of the evaluation of clone detection software at the *First International Workshop on the Detection of Software Clones.*

In the evaluation of Abyss, the clone detector found five groups of duplicate functions. However, only two are related to Abyss, while the rest are duplicate functions in predefined header files. The first group related to Abyss consists of the functions *ConfGetToken* (in conf.c) and *GetToken* (in http.c). These two functions represent 23 lines of code that are exact matches of each other. The second group consists of the functions *ThreadRun* (in thread.c) and *ThreadStop* (in thread.c). These two functions represent 5 lines of code that call different functions in their return values, but have similar return types.

In the evaluation of Weltab, the clone detector found six groups of duplicate functions. Two are related to duplicate functions in predefined header files. The remaining four groups are functions scattered in different files where only their "main" functions differ. The following lists the groups of clones excluding the ones found in the predefined header files.

Group No. 1:	Function *canvw* in files canv.c, cnv1.c, and cnv1a.c
Group No. 2:	Function *lhead* in files lans.c and lansxx.c and function *rshead* in files r01tmp.c, r101tmp.c, r11tmp.c, r26tmp.c, r51tmp.c, rsum.c, and rsumxx.c
Group No. 3:	Function *rsprtpag* in files r01tmp.c, r101tmp.c, r11tmp.c, r26tmp.c, r51tmp.c, and rsum.c
Group No. 4:	Function *askchange* in files vedt.c, vfix.c, and xfix.c

Initially, the second category of clones was not allowed to have different types. By relaxing this requirement on the second step of the detection process, three additional clone groups were found.

[2] http://abyss.sourceforge.net/

[3] http://www.iste.uni-stuttgart.de/ps/clones/

These groups contained pairs of duplicate functions that dealt with the conversion of two types of values: `int` and `long int`. The pairs were all found in baselib.c and are functions *cvci* and *cvcil*, functions *cvic* and *cvicl*, and functions *cvicz* and *cviczl*.

5. RELATED WORK

Several clone detection methods have used the AST representation of a program to find clones [2][5][7][10]. Generally, a clone detection tool uses an AST that is generated by a pre-existing parser. The advantage of Phoenix is that it provides a framework where customized software analysis tools (e.g., clone detection) can be added or plugged in. Because Phoenix serves as a platform for compiler development, AST generation is already part of the frontend.

Baker describes one of the earliest applications of suffix trees to the clone detection process [1]. However, instead of AST nodes, a token-like structure produced after the lexical analysis is used to find duplicates. An AST abstracts much of these tokens, while preserving the structure of the program. The combination of ASTs and suffix trees to find code clones is unique to our approach.

The utilization of biological sequence matching algorithms is evident in [7] and [10]. Both use string alignment algorithms that incorporate dynamic programming methods. This method is useful in the detection of near exact clones. Although suffix trees are not effective in the detection of near exact clones, they play a role in a hybrid dynamic programming method that reduces the time to perform dynamic programming calculations. The use of suffix trees in this hybrid dynamic programming method is discussed as future work in the next section.

6. CONCLUSION

We have introduced a clone detection technique that finds exact matching functions by performing searches on suffix trees generated from a program's AST representation. The implementation of this technique plugs into Microsoft Phoenix's backend process. Further enhancements can expand the types of clones that can be found. The remainder of the conclusion describes several areas for future work.

Suffix trees are not effective when searching for near exact matches. Algorithms that offer better results for near exact matches include the Smith-Waterman algorithm (local sequence alignment), as used in [7]. This algorithm utilizes a dynamic programming table to determine the most optimal alignment between two strings. The calculations of a dynamic programming table consists of computing the values of each cell in an n x m table, where n and m are the respective lengths of the two strings that are being compared. Depending on the length of the strings, the computation time of the table values can be exponential. A method called *k-difference inexact matching* can reduce the amount of calculation needed on the dynamic programming table [6]. This is done by using a hybrid dynamic programming process that utilizes suffix trees. In effect, suffix trees become part of the method to find near exact matches. The continuation of the work from this paper is to develop an implementation of the *k-difference inexact matching* method to search for near exact functions.

This paper focused on clones at the function-level. Code clones can occur at several different levels of granularity (e.g., from the statement level to the program level). Statement-level clones can reveal operations that suggest a crosscutting concern, which could be made into an aspect. Clones at the program level can represent entire programs that are clones of one another. The detection of program-level clones may be useful to check for duplicate submissions of homework in a programming class, such as the approach adopted by the popular web-based program MOSS (A Measure of Software Similarity)[4]. A more robust clone detector that can perform evaluations on multiple levels of granularity would enhance the benefit of a clone detection tool.

Currently, our clone detection tool recognizes AST nodes for the C language. We want to expand the coverage of the tool to other languages, such as C++ and C#. This requires the tool to have knowledge of the AST nodes for additional languages. Another approach is to develop a language-independent technique to clone detection, which would reduce the challenge of updating the tool for each new language to be supported.

Detecting code clones can be done by evaluating suffix trees generated from the nodes of an abstract syntax tree. Development in Microsoft's Phoenix provides a supportive framework for the clone detection tool. Further development of the tool will allow it to detect more types of clones, both in terms of structure and in terms of granularity, in addition to expanding its language base.

A project website (http://www.cis.uab.edu/tairasr/clones) for our clone detector contains general information and a video demonstration.

7. ACKNOWLEDGEMENTS

We thank Dr. Alan Sprague for his feedback regarding the use of biological sequence algorithms in code clone detection.

8. REFERENCES

[1] Baker, B. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proceedings of the Second IEEE Working Conference on Reverse Engineering*, Toronto, Canada, July 1995, pp. 86-95.

[2] Baxter, I., Yahin, A., Moura, L., Sant'Anna, M., and Bier, L. Clone Detection using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance*, Bethesda, MD, November 1998, pp. 368-377.

[3] Bruntink, M., van Deursen, A., van Engelen, R., and Tourwé, T. On the Use of Clone Detection for Identifying Crosscutting Concern Code, *IEEE Transactions on Software Engineering*, vol. 31, no. 10, October 2005, pp. 804-818.

[4] Delcher, A., Phillippy, A., Carlton, J., and Salzberg, S. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Research*, vol. 30, no. 11, June 2002, pp. 2478-2483.

[5] Evans, W. and Fraser, C. *Clone Detection via Structural Abstraction*. Technical Report MSR-TR-2005-104, Microsoft Research, Redmond, WA, 2005.

[6] Gusfield, D. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, New York, NY, 1997.

[4] http://www.cs.berkeley.edu/~moss/general/moss.html

[7] Greenan, K. Method-Level Code Clone Detection on Transformed Abstract Syntax Trees using Sequence Matching Algorithms. Student Report, University of California - Santa Cruz, Winter 2005, available at http://www.cs.ucsc.edu/~ejw/courses/290gw05/greenan-report.pdf.

[8] Höhl, M., Kurtz, S., and Ohlebusch, E. Efficient multiple genome alignment. In *Proceedings of the Tenth International Conference on Intelligent Systems for Molecular Biology*. Supplement of Bioinformatics, Edmonton, Canada, August 2002, pp. 312-320.

[9] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J-M., and Irwin, J. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*. Springer-Verlag LNCS 1241, Jyväskylä, Finland, June 1997, pp. 220-242.

[10] Kontogiannis, K. Pattern Matching for Clone and Concept Detection. *Automated Software Engineering*, vol. 3, nos. 1-2, July 1996, pp. 77-180.

[11] Microsoft Phoenix, http://research.microsoft.com/phoenix

[12] Tarr, P., Ossher, H., Harrison, W., and Sutton, Jr., S. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999, pp. 107-119.