Eötvös Loránd University
Faculty of Informatics
Department of Programming Languages and Compilers

Text duplication analysis using suffix trees

Supervisor:
Zoltán Porkoláb
Associate Professor

Author:
Balázs Kezes
Computer Science MSc

Budapest, 2012

Contents

1 Introduction **3**

2 Applications **4**

2.1 Source code analysis 4

2.1.1 Introduction 4

2.1.2 Reason of the duplicates 5

2.2 Plagiarism 5

2.3 Redundancy removal 6

3 Related work **7**

3.1 Source code analysis 7

3.2 Plagiarism 8

4 Implementation **10**

4.1 Code structure 10

4.2 Duplicate detection engine 11

4.2.1 User interface 11

4.2.2 Algorithm 14

4.3 Input transformers 20

4.3.1 The text transformer 21

4.3.2 The source code transformer 21

4.4 Output presenters 21

4.4.1 The raw presenter 21

4.4.2 The text presenter 22

4.4.3 The source code presenter 22

4.5 Hardware and software requirements 23

5 Evaluation **24**

5.1 Time and memory performance 24

5.1.1 Data used 24

5.1.2 Results 24

5.2 Real world results 27

5.2.1 The complete works of Shakespeare 27
5.2.2 The complete works of Charles Dickens 28
5.2.3 The Holy Bible 30
5.2.4 Linux kernel version 3.3 source code 30
5.2.5 rxvt-unicode terminal emulator version 9.14 source code . . . 32

6 Conclusion **34**
6.1 Summary 34
6.2 Further development possibilities 34

Bibliography **35**

A CD-ROM contents **38**

Chapter 1

Introduction

The need for searching duplications arises in many areas of computer science. One can find examples in plagiarism detection, source code analysis, compression algorithms, DNA analysis and many others.

This document uses the word "duplicate" primarily to describe a piece of continuous text which appears at multiple distinct places in one or multiple text documents. There are other synonyms like copies, clones, replicas, repetitions and many others. In this context they all mean the same and we use these words interchangeably. Usually there is one specific word used in a specific application field.

This thesis summarizes the duplication analysis in various fields, describes the methods used there and finally gives a generalized tool which can be configured to be used in many different situations. This generalized tool uses suffix trees as its primary data structure for searching the duplicates in the documents.

This document is organized as follows. Chapter 1 gives a quick introduction to the area of duplication search and describes the structure of this work. Chapter 2 gives an overview of various computer science areas where there is a need for duplication search and describes how is it used in that particular area. Chapter 3 walks through the common solutions to this problem and discusses their drawbacks, advantages and performance characteristics. Chapter 4 gives an overview of the thesis' solution to the problem and its implementation specifics, nuances and the solution's hardware and software requirements. Chapter 5 evaluates this solution on various test inputs and measures the solution's memory and time behaviour. Chapter 6 summarizes the results published in this thesis.

Chapter 2

Applications

2.1 Source code analysis

2.1.1 Introduction

In computer programming the duplications are those sections of code which occur more than once within a program or across different programs. In this area the duplications are commonly considered undesirable for a number of reasons. There are different ways in which two sequences of code can be considered a duplicate such as:

- exact byte-to-byte duplicate
- exact byte-to-byte duplicate but the whitespace and comments are ignored
- same code structure (the variables used can be renamed)
- like one of the above but allow slight variations of insertion, deletion and modification of some bytes and tokens
- same functionality - the code does the same but it is written in a different way (this is very hard to detect automatically and this thesis does not discuss this further)

Current literature considers code duplicates to be harmful [1, 4, 6] although there are some upsides and advantages of them based on the reason of their creation. The downsides include the following:

- It makes the code's intention harder to understand, because it is not clear what is different between two similar code sections.
- Their maintenance is a burden because the same change should be applied to multiple places. This can be easily forgotten if the developer is not aware of the existence of the duplicates in the code base.

The code duplicates can be usually removed by abstracting the common code into a class or function and just parameterize that as needed.

2.1.2 Reason of the duplicates

There are legitimate reasons where code duplicates have a well understood reason and they are a much better alternative than abstracting out the common parts. These arises of duplications in a code base were well researched by the literature and these duplications can be categorized into a few categories [2].

- Forking: The forking pattern involves copying a larger portion of code with the intention that the resulting duplicate will evolve on its own. This pattern encompasses three subcategories:
  - Hardware variations
  - Platform variations
  - Experimental variations
- Templating: The templating pattern occurs when an existing code closely satisfies a given need. Often templating is used as a form of parametrization. This pattern has also three subcategories:
  - Boiler-plating due to language in-expressiveness
  - API/Library protocols
  - General language or algorithmic idioms
- Customization: This arises when a code solving a very similar problem already exists but additional or differing requirements create the need for extension or modification. This pattern can be subdivided into two subcategories:
  - Bug workarounds
  - Replicate and specialize

2.2 Plagiarism

Plagiarism is a special case of duplication where parts of various original works were copied into a document without giving appropriate citations. The plagiarism affects many areas of life like academy, journalism, web content, arts and even politics.

There are many forms of plagiarism where the author doesn't give citation:

- word for word copy of other's work
- copying of large amounts of work with small alterations
- paraphrasing other works

Even if the author gives citations there may be still problems with his work:

- the citation is not specific enough so it can't be easily found
- the author cites inaccurately

- the whole work is just a citation and doesn't contain original work

Detecting and eliminating plagiarism is important because if not addressed then the work done by the original authors will be undermined and credit will be wrongfully given to the copycat. By not giving credit to the original authors of the work their fame falls and this can result in cutting the funding of people actually doing the research. The problem with this kind of plagiarism is that the original work can be anything: a journal article, an article on a website, a transcription of a speech, etc for which huge databases are needed. Usually companies specializing in plagiarism detection are constantly indexing the internet and updating their huge databases.

A rather more controlled place where plagiarism occurs are the school essays and homeworks. These assignments can be finely specialized so that finding works to copy for these is very hard. The only source of plagiarism is student collaboration or plagiarism of the previous years' works. But as these are turned in, these can be used to build a smaller database for which a simpler plagiarism detection tool suffices.

2.3 Redundancy removal

Duplication detection is also used for finding redundancies in complex databases which can be then eliminated and memory can be then saved. A typical example is a mail server. An email can be received by many people. For example somebody sends a message with a 2MB attachment to 1000 people. Now this attachment will duplicated 1000 times and will take 2GB of the mail server's space. A duplication detection engine can find these and replace each copy of the image with a reference to a unique copy which will free up wasted space.

Chapter 3

Related work

3.1 Source code analysis

Baker's algorithm

Baker's algorithm was first implemented in the program called *Dup* by Brenda Baker [3]. This algorithm begins by stripping whitespace and comments from the source files. Then it hashes the lines so that "similar" lines get the same hash. After this step we get a list of hashes in which we need to find duplicate sections. The original paper describes two methods for this step. The first is an algorithm with a running time of $O(n+p)$ where n is the number of lines and p is the number of distinct pairs of identical lines. The second approach uses suffix trees and has $O(n+m)$ running time where m is the number of matches found.

In order to find renamed variables the method replaces the tokens with a 'P' before running the exact matching algorithms. After the matching algorithm returns the duplicates the method analyses them whether the renaming was consistent or not. This approach is very similar to the approach used in this thesis so these will be further discussed in more detail later in chapter 4.

Clone detection using abstract syntax trees

This approach parses the source code into its AST form in which the duplicates are sought. If the number of nodes in AST is n then the brute force algorithm runs in $O(n^3)$: comparing two subtrees is $O(n)$ and there are $O(n^2)$ pairs of subtrees needed to be compared. An algorithm by Ira Baxter et al. [4] reduces this to $O(n)$ by having a fast subtree comparison algorithm which runs in constant time on average, and by hashing the subtrees. This way the number of possible pairs of subtrees to be compared is significantly reduced.

The method in the cited article also supports ignoring the names of the variables just by ignoring the leaf nodes of the AST.

Visual Detection of Duplicated Code

The utility called *Duploc* by Rieger and Ducasse [5] begins by *normalizing* the input text lines. The lines are represented in a huge 2-dimensional matrix view where columns and rows represent the lines. If an element of the matrix is represented by a black dot then it means that the given row and column represent a duplicate. This allows us to visually inspect the amount of duplication in different parts of the code.

3.2 Plagiarism

Fingerprinting

The method of fingerprinting as described in [7] is based on the following steps:

- select a set of subsections of the document
- hash each subsection to create the *minutiae*
- look these minutiae up in a database for matches
- store a subset of the minutiae in the database to represent the document

The fingerprinting's accuracy hugely depends on its parameters, like

- which subsections to select for hashing
- what hash function to choose
- what makes up the minutiae of an entry in a database

If these parameters are chosen wisely then the fingerprinting methods work quite well even for large number of documents.

Classical string matching

These methods look for exact duplicates in the documents. Usually suffix trees or suffix vectors are used. These methods don't quite work for large number of documents because of the overhead of the complex data structures.

Bag of words analysis

This method first strips the documents from pronouns, prepositions, conjunctions and other not important words. Then the method converts the remaining words into a canonical form which includes removing the "-ful" suffix from the word, converting the plural forms into singular and similar. The next step is to select the most common words from the document. These will be called *keywords* of the document.

To compare two documents the method of *cosine similarity measure* is used. Let the dimensions be the keywords of the two documents, and let's assign a vector to each document. The magnitude of a specific dimension is the number of times the given word occurred in the document. Then the similarity between the two documents is the following:

$$S(A,B) = \frac{\sum_{i=1}^{n} A_i \cdot B_i}{\sqrt{\sum_{i=1}^{n} A_i^2 + \sum_{i=1}^{n} B_i^2}}$$

where n is the number of keywords. A value near to 0 means that they are similar and a value near to 1 means that the two documents are different.

An example implementation for such system is *CHECK* described in [8].

Chapter 4

Implementation

4.1 Code structure

The application created by this solution is named DDD which is an acronym for Data Duplication Detector.

The solution is written in pure C targeted at Linux 3 and it code's directory structure consists of the following directories:

- `/`
- `/src/`
- `/src/common/`
- `/build/`
- `/build/common/`

The root directory contains a README file which contains this same information. The Makefile for the solution is also in the root directory. The directories beginning with build will contain all binaries created via the Makefile from the same directory beginning with src.

The solution will consists of multiple binaries. The common parts of the code are residing in the `/src/common` directory. The source files in this directory are compiled to a shared library which is then linked to all executables generated by this solution.

The executables' name have the form of `ddd-[type]-[name]` where `[type]` can be one of the following:

- `eng`: Represents an duplication finding "engine". Currently there is only one but during the development there were several other experimental versions from which the good parts were incorporated into the main engine.
- `tr`: Input transformer. The executables take a specific form of input file and transform them so that they can be used for exact duplicate searching. For

example `ddd-tr-text` removes superfluous whitespace and punctuation and transforms each letter to lowercase.

- `pr`: Output presenter. This executable takes the files generated by the input transformer, the results generated from the engine and presents the results in a human readable way for example with static HTML files.

The following executables will be generated:

- `ddd-eng-fasttree`
- `ddd-tr-text`
- `ddd-tr-syntax`
- `ddd-pr-htmlraw`
- `ddd-pr-htmltext`
- `ddd-pr-htmlsyntax`

Each application is described in more detail in the following sections.

4.2 Duplicate detection engine

4.2.1 User interface

The `ddd-eng-fasttree` is a simple command line driven application where the user issues one of the available commands and observers its results.

The following commands are available:

- exit: quits
- help: this help
- dump_dot [*filename*]: dump the suffix tree into *filename* in dot format
- load [*filename*]: loads the given *filename*
- min_matches: minimum number of duplicates in a result [default 2]
- min_length: minimum length of a result [default 2]
- match_count: print the number of unique duplicates and the number of entries to be reported in a run
- run [*filename*]: run the currently configured query and optionally dump the result into *filename* instead of standard output
- stats: display some statistics about the data structures

Example usage:

```
cd linux-2.6/kernel
$ ddd-eng-fastree
```

Now let's load some files:

```
load auditfilter.c
    auditfilter.c done [  34129 bytes]
load audit_watch.c
    audit_watch.c done [  14515 bytes]
load printk.c
    printk.c done [  44553 bytes]
```

We can query some statistics about the internal statistics (details about what these mean is described in the next subsection):

```
stats
    character count:            93197
    node count:                147351
    memory reserved:             1610 MB
    memory used:                   97 MB
    memory forever wasted:          1 MB
```

We order the engine to look for matches which occur at least at two places:

```
min_matches 2
    min_matches = 2
```

We now order the engine to look for matches with at least 500 characters in length:

```
min_length 500
    min_length = 500
```

We can now query the engine how much results would it return if we would run it:

```
match_count
    472 944
```

This means it can find 944 matches which encompass the 472 unique substrings. We can increase the minimum length in order to decrease the number of matches:

```
min_length 970
    min_length = 970
match_count
    2 4
```

We can now run the engine:

```
run
    2 970
    971 2
        auditfilter.c 71
        audit_watch.c 61
    970 2
        auditfilter.c 72
        audit_watch.c 62
    0 0
```

The first line echoes the input parameters min_length and min_matches and the rest are the duplicates found. Here only one duplicate is, found although it is found twice because both match the input specifications of duplicates equal or larger than 970. We can exit the engine and run a presenter now to examine the results:

```
    exit
$ ddd-pr-ttyraw
```

We enter a result (more can be specified, we specify only one):

```
    971 2
        auditfilter.c 71
        audit_watch.c 61
    0 0
```

The presenter will output the duplicate surrounded with some context lines. In this particular example the following lines are printed (colors and long lines are removed for technical reasons):

```
/* audit_watch.c -- watchin | /* auditfilter.c -- filteri
 *                         |  *
 * Copyright 2003-2009 Red  |  * Copyright 2003-2004 Red
 * Copyright 2005 Hewlett-P |  * Copyright 2005 Hewlett-P
 * Copyright 2005 IBM Corpo |  * Copyright 2005 IBM Corpo
 *                         |  *
 * This program is free sof |  * This program is free sof
 * it under the terms of th |  * it under the terms of th
 * the Free Software Founda |  * the Free Software Founda
 * (at your option) any lat |  * (at your option) any lat
 *                         |  *
 * This program is distribu |  * This program is distribu
 * but WITHOUT ANY WARRANTY |  * but WITHOUT ANY WARRANTY
 * MERCHANTABILITY or FITNE |  * MERCHANTABILITY or FITNE
 * GNU General Public Licen |  * GNU General Public Licen
 *                         |  *
 * You should have received |  * You should have received
 * along with this program; |  * along with this program;
 * Foundation, Inc., 59 Tem |  * Foundation, Inc., 59 Tem
 */                         |  */
                            |
#include <linux/kernel.h>   | #include <linux/kernel.h>
#include <linux/audit.h>    | #include <linux/audit.h>
#include <linux/kthread.h>  | #include <linux/kthread.h>
#include <linux/mutex.h>    | #include <linux/mutex.h>
```

```
#include <linux/fs.h>        | #include <linux/fs.h>
#include <linux/fsnotify_ba | #include <linux/namei.h>
#include <linux/namei.h>     | #include <linux/netlink.h>
#include <linux/netlink.h>   | #include <linux/sched.h>
```

The above demonstration shows an example where we are searching for exact duplicate. Examples for non-exact matches can be found in the presenter section.

4.2.2 Algorithm

A suffix tree is a special radix tree which holds all the suffixes of the strings contained in it. We can get the string represented by a node by concatenating the edge labels on the path from the root to the node. A label on an edge to a child can contain a string with one or more letters. Each children label must begin with a different letter. An edge which contains more than one letter is usually named as a compressed edge. Current literature doesn't allow nodes with one child but we will allow them in one special case: if the node represents the last character of a string then it can have one child. Each node in a suffix tree has a suffix link which points to another node which represents the strict suffix of the current node.

Let's see an example which clarifies all these concepts. Let's assume we loaded the file called `f` which contained the string `banana`. This means that the resulting tree must contain the strings `banana`, `anana`, `nana`, `ana`, `na` and `a`. The resulting suffix tree is the following:

> The file `banana.pdf` hasn't been created from `banana.dot` yet.
> We attempted to create it with:
> '`dot -Tps2 banana.dot | epstopdf --filter -o=banana.pdf`'
> but that seems not to have worked. You need to execute '`pdflatex`' with
> the '`-shell-escape` option. You also need '`epstopdf`' from CTAN.

Figure 4.1: The suffix tree of "banana"

Here each node is also an end node. The suffix links are the dashed edges. For example node 2 represents the string banana, its suffix link points to 3 which represents the string anana.

Let's see an example with two files, `f` with `banana` and `g` with `bandana`. The suffix links are not displayed now as they would make the graph too complicated:

The file bananabandana.pdf hasn't been created from bananabandana.dot yet.
We attempted to create it with:
'dot -Tps2 bananabandana.dot | epstopdf --filter -o=bananabandana.pd
but that seems not to have worked. You need to execute 'pdflatex' with
the '-shell-escape option. You also need 'epstopdf' from CTAN.

Figure 4.2: A suffix tree containing both "banana" and "bandana" words

The algorithm described in this chapter is a heavily modified algorithm based on Ukkonen's original linear time algorithm [9]. These modifications are needed to lift some of the restrictions the original paper had, namely that the string must be ending in a unique character and that the suffix tree represents only one string.

Data structures

There are some key data structures needed to be known in order to understand the engine's suffix tree solution. Every loaded file's name and contents is stored in the program's memory. Internally a file is represented by an integer. A node in the tree can represent multiple files so we have a structure for a list of files:

```
struct filelist {
    int file;
    struct filelist *next;
};
```

In order to represent a tree we will need a structure representing for a node and a structure for representing an edge to a children:

```
struct child_desc {
    int node;
    int file;
    int start, end;
};

struct node {
    struct filelist *files;
    int parent;
    int suffix;
    int children_count;
    struct child_desc children[0];
};
```

The node ID is a one-based index so the zero can be used for unitialized variables. The `child_desc` structure's `node` member points to the child node represented by this edge. The `(file, start, end)` triplet represents the string on the edge. The file's contents are in the memory and an edge's label can always represented by the continous bytes of a file.

The `files` points to a list of files for which the current node represents a suffix. This is what the right box contains in the graphs above. The `parent` and `suffix` point to another nodes suggested by their names. The node with ID of 1 also known as the root node doesn't have a parent so its parent is a zero. The `children_count` is the size of `child_desc` array which is followed after this member. This can be bigger than the actual number of children, the unused `child_desc` entries are marked by pointing their `node` member to zero and these unused entries are at the end of the array forming a continous sequence.

It can be shown that such tree consists less than or equal $2n$ nodes where n is the number of bytes in all files [9].

For fast allocation of nodes a big pool is preallocated at the start of the program. It doesn't support resize so care must be taken that the initial allocation is big enough. Let n be the number of bytes in all files then $300n$ is an optimistic upper bound for the recommended size of the preallocated pool. Note that this pool is not only used for nodes but also used for other allocations so it is not easy to determine an exact upper bound.

Adding a string to the tree

In order to demonstrate the algorithm used we will walk through an example. We will be adding the string `bandana` to a tree which already contains the string `banana`.

In these graphs the middle part of a node represents the suffix node. It is showed this way because the arrows would make the graphs unreadable.

The file `streestruct.pdf` hasn't been created from `streestruct.dot` yet.
We attempted to create it with:
'`dot -Tps2 streestruct.dot | epstopdf --filter -o=streestruct.pdf`'
but that seems not to have worked. You need to execute '`pdflatex`' with
the '`-shell-escape` option. You also need '`epstopdf`' from CTAN.

Figure 4.3: The structure of a node which will be used in the next figures

The starting point is the following:

The file banana3.pdf hasn't been created from banana3.dot yet.
We attempted to create it with:
'dot -Tps2 banana3.dot | epstopdf --filter -o=banana3.pdf'
but that seems not to have worked. You need to execute 'pdflatex' with
the '-shell-escape option. You also need 'epstopdf' from CTAN.

Figure 4.5: The edge (1-2, "banana") must be splitted in order to add "bandana".

The file banana2.pdf hasn't been created from banana2.dot yet.
We attempted to create it with:
'dot -Tps2 banana2.dot | epstopdf --filter -o=banana2.pdf'
but that seems not to have worked. You need to execute 'pdflatex' with
the '-shell-escape option. You also need 'epstopdf' from CTAN.

Figure 4.4: The starting suffix tree which contains only the word "banana". The word "bandana" will be added to this tree.

As the next step we add the string bandana but while adding this string we need to ensure that the suffix links are maintained properly. If we can do this we are almost ready with the new tree.

As the first step we check that the word bandana is already in the tree. If yes, we are ready. If not it means that we will need to split an edge somewhere and possibly add a new node. In this case the edge $1 \rightarrow 2$ must be splitted into ban and ana edges:

As you can see a new node was allocated with the ID of 8. Notice that the IDs of already existing nodes don't change. If a given string is represented by a given ID, then it will be so until the end of the program. The suffix link for this new 8th node is not known so we have to fix that up. For that we have original edge's source node and traverse its suffix with the new edge's label. The new edge's label is ban but in this case we have to traverse an because the source edge was the root (this is the only special case). But a node for an doesn't exist from the root node so we have to split node 5. This will create node 9 which will be the 8th node's suffix. But after we created node 9, we have to fix up its suffix too! To fix that up we will need to split the edge $1 \rightarrow 6$. This will create node 10 and because it is represented by a one length string its suffix is trivially the root. After the fixups we still need to add a new child to the node 8 with the label of dana. This will point to node 11 and will represent the string bandana.

Note here that the number of *fixups* is strictly less than n where n is the length of the string added. This is because each time we *fix up* a node the new node represents a suffix of the string added and there are less than n such suffixes. Also note that

the number of nodes visited during the *fixups* will be linear because when we pass an edge, then the bytes on that edge won't be needed to be considered again because we will use the edge target's suffix for further *fixups*. Furthermore, because we are splitting nodes, which were already in the tree, we can make the string comparisons between the edge labels and string to be traversed to $O(1)$ just by noticing that the edge labels in a node and its suffix node consists of the same strings. For an example look at the above graph's 6th node. The 6th node's child edge `na` is the same as its suffix node's, 7th node's, `na`. Length can be the only difference here. These observations make this step $O(n)$.

After the above step we get the following graph:

The file `banana4.pdf` hasn't been created from `banana4.dot` yet.
We attempted to create it with:
'`dot -Tps2 banana4.dot | epstopdf --filter -o=banana4.pdf`'
but that seems not to have worked. You need to execute '`pdflatex`' with
the '`-shell-escape` option. You also need '`epstopdf`' from CTAN.

Figure 4.6: Here the edge (7-5, "na") is splitted in order to prepare for inserting "andana".

After the previous step it is still possible that a new node which represents the newly added string doesn't have a suffix link. In this case the new node is a leaf node and we have to fix that up as well. This is very similar to adding a new string to the tree (`andana` in this case) but instead of starting at the root we start from the new node's parent's suffix node. The new node is 11, its parent node is 8 whose suffix node is 9 which represents the string `an`. All we need to do is to add the string `dana` to the subtree rooted at 9. This will create a new node 12 and a new edge $9 \rightarrow 12$ labeled `dana`. Similarly we will create a new edge $10 \rightarrow 13$ where node 13 will represent the string `ndana`. Node 14 and the edge $1 \rightarrow 14$ representing the string `dana` will also be created. After this step we will find that the string `ana` is already in the tree and so we don't need to continue this process.

Similar analysis as in the previous step will show us that this step is also $O(n)$.

The resulting tree is following:

The file banana5.pdf hasn't been created from banana5.dot yet.
We attempted to create it with:
'dot -Tps2 banana5.dot | epstopdf --filter -o=banana5.pdf'
but that seems not to have worked. You need to execute 'pdflatex' with
the '-shell-escape option. You also need 'epstopdf' from CTAN.

Figure 4.7: The suffix tree after the string "bandana" was inserted but with the marks in the third field missing.

As you can see we are almost ready. The only thing missing is to mark the nodes with the file they are representing. This means we have to mark up bandana and all of its suffixes with the file g. In order to do this we will traverse all the suffix nodes through the suffix links in the tree. In the current case we start at node 11 and then visit the following nodes: 12, 13, 14, 5, 6, 7. We add file g to all of these nodes. This step is also $O(n)$. After this step we are finished and the algorithm runs in $O(n)$. The resulting tree is the same as figure 4.2.

Running a query

We specify only one query on the tree: return all duplicates which have at least a minimum user defined length and have at least a minimum user defined number of matches. The answer to this query are the strings of the defined minimum length which are represented by nodes which have at least as many *leafs* originating from it as many matches are needed. Note that here we use the term *leaf* specially. A *leaf* is a node which represents a suffix of a file. A node can represent multiple leafs if it represents suffixes of multiple files. For example node 5 in the banana-bandana tree in figure 4.2 represents the suffix ana which is suffix of both strings so it counts as two leafs. The subtree rooted at node 9 has 4 leafs in total in this regard.

To determine these nodes we have to do a depth first search on the tree. A simple recursive approach is not sufficient because the tree can have very deep nodes which would result in stack overflow. An alternative approach would be to manage the stack manually or to add some new data members to the nodes which would aid in the traversal. This solution chose the latter approach.

We add the following structure to the node's structure:

```
struct dfs_data {
    int next_child;
    int val1, val2, val3;
};
```

The `val1`, `val2`, `val3` members are general purpose variables whose purpose is depending on the actual DFS. The member `next_child` is to mark the next child to be processed after a child has been processed. If this points beyond the last child it means that after the current child node is processed and we finish processing the current node we can return to the parent node.

We can look at DFS as a loop which takes a function which takes a node identifier and returns another node identifier or 0 to report finish. The C code is as simple as it sounds:

```
typedef int(*dfs_func)(int);

void dfs_loop(dfs_func f)
{
    int node = 1;
    while (node != 0) {
        node = f(node);
    }
}
```

The `dfs_func` which finds the duplicates works by the following way: after a node is fully processed `val1` stores the length of the string represented by the node and `val2` stores the number of *leafs* in the subtree rooted in the node. If in a node these two variables meet the constraints for a match then the node is marked for printing. Then another DFS goes through the tree and prints each *leaf* which is below a node marked for printing.

4.3 Input transformers

Currently there are two transformers. Each of these take an input a generate three outputs. These three outputs and the output of the duplication detection engine must be passed to the presenter in order to present the results. This is not universal so new transformers could be written with different rules.

The three outputs:

- copy: this a raw copy of the input, this will be used to represent the original input when presenting (this is useful when the input is coming from a pipe for example)
- reduced dump: this file will contain the processed input suitable for finding matches in it so this is passed to the duplication detection engine
- presenter data: this file will contain the data needed to reconstruct a substring in the original input from a substring in the reduced dump

4.3.1 The text transformer

The text transformer removes all superfluous whitespace and punctuation and converts all letters to lower case.

Example input:

```
To be, or not to be, that is the question:
Whether 'tis Nobler in the mind to suffer
```

Example output:

```
to be or not to be that is the question whether tis
nobler in the mind to suffer
```

The output doesn't contain newlines; the example output contains them for formatting reasons.

4.3.2 The source code transformer

The source code transformer removes all whitespace, replaces each word except keywords with I, strings with S, single character quotes with Q and numbers with N. Keywords and operators are left untouched.

Example input:

```
int main()
{
  if ('\n' == 10) {
    printf("return world!\n");
  }
  return 0;
}
```

Example output:

```
II(){if(Q==N){I(S);returnN;}
```

4.4 Output presenters

4.4.1 The raw presenter

The raw presenter named `ddd-pr-htmlraw` generates HTML documents representing the results. This application must be run from the same directory as the duplication engine. The HTML files are generated in the current directory. Here's

an example HTML page generated from Shakespeare's works (markup and colors omitted):

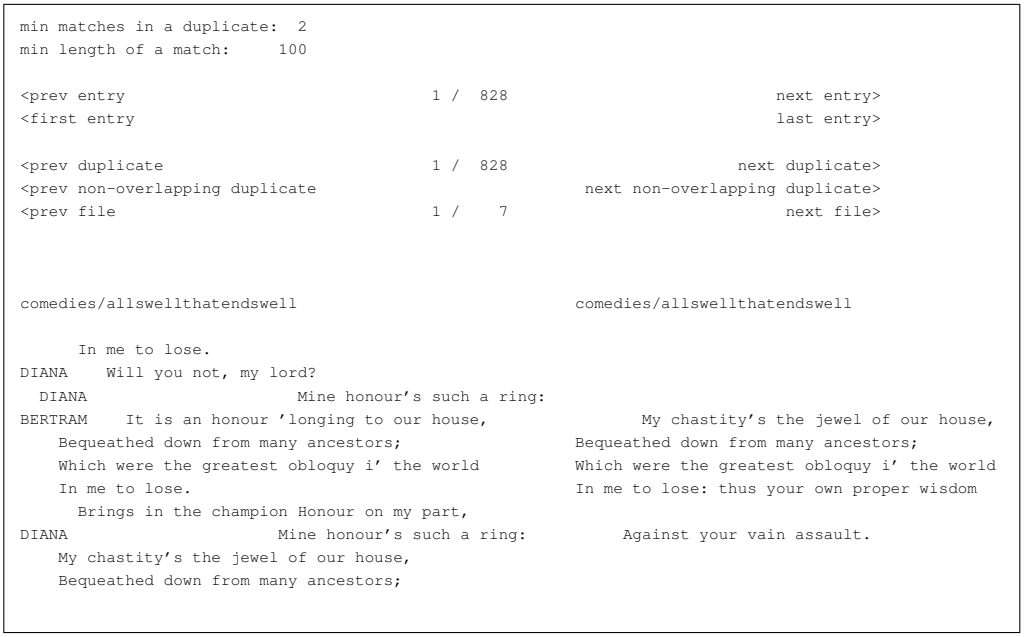

Figure 4.8: A sample from the generated HTML files

4.4.2 The text presenter

The text presenter is very similar to the raw presenter but it correctly transforms the beginning and the end of the selection into a selection from the original input with the help of the presenter data.

4.4.3 The source code presenter

The source presenter does all what the text presenter does but it has one additional feature. It checks whether each identifier in the duplication is consistently *renamed*. For example, let's suppose we have these two input texts: "`apple banana apple banana`" and "`pear banana pear orange`". Both get transformed to `IIII` and `IIII`. Let's assume the duplication detection engine found these 4 characters as a duplicate. The first character represents `apple` in the first text, `pear` in the second. The second character represents `banana` in both texts. The third character also represents `apple` and `pear`. This *renaming* was checked for consistency which means that the previous `apple` *renaming* was checked whether it was also `pear`. It was; so this *renaming* is consistent. The fourth character is `banana` and `orange`. This is also checked for consistency but this fails because previously `banana` was mapped to `banana` so this case is marked as inconsistent.

Inconsistencies are marked with red in the generated HTML files.

4.5 Hardware and software requirements

Hardware needed:

- x86_64 or compatible processor
- 1 MB of disk space for the binaries
- 512 MB + additional memory depending on the input (300n bytes where n is the input size)

Software needed:

- gcc version 4 or better (for compilation)
- GNU make version 3 or better (for compilation)
- Linux version 3 or better
- GNU libc 2 or better

Chapter 5

Evaluation

All performance measurements in this section were done on a Core 2 Duo T6400 running at 1.2 GHz with 2048 KB cache.

5.1 Time and memory performance

5.1.1 Data used

Three sets of data is used to test various performance characteristics. The first data consists only of the letter `a`. This will create a suffix tree with the least number of nodes. The second data consists of random files which gives an unpredictable number of nodes. The third data consists only of the letter `a` but with a newline appended. This constitutes as the worst case because this will create the most number of nodes in the suffix tree because in this case two nodes are needed for each letter except the last two as the following figure illustrates:

The file `worstcase.pdf` hasn't been created from `worstcase.dot` yet.
We attempted to create it with:
'`dot -Tps2 worstcase.dot | epstopdf --filter -o=worstcase.pdf`'
but that seems not to have worked. You need to execute '`pdflatex`' with
the '`-shell-escape` option. You also need '`epstopdf`' from CTAN.

Figure 5.1: The suffix tree of "aaaaab". All letters are the same except the last constitute as the worst case in the following measurements.

5.1.2 Results

The results are shown in graphs. The results are quite straightforward once the effects of the processor's cache are factored in. Figure 5.2 shows that even though

random data input has much less nodes than the max nodes input it is still much-much slower. This is because the max nodes input structure's memory access pattern is very easily predictable for the cache. The effects of the cache are best shown in Figure 5.4. To run a query in the min nodes input or the max nodes input is very fast. In the case of random data input the execution time grows very fast until the 2 MB cache is fully depleted and once the application is fully bounded by the main memory's speed then the time continues to grow very slowly.

The application was checked for memory leaks and other problems with the utility Valgrind [10] and no errors have been found in it.

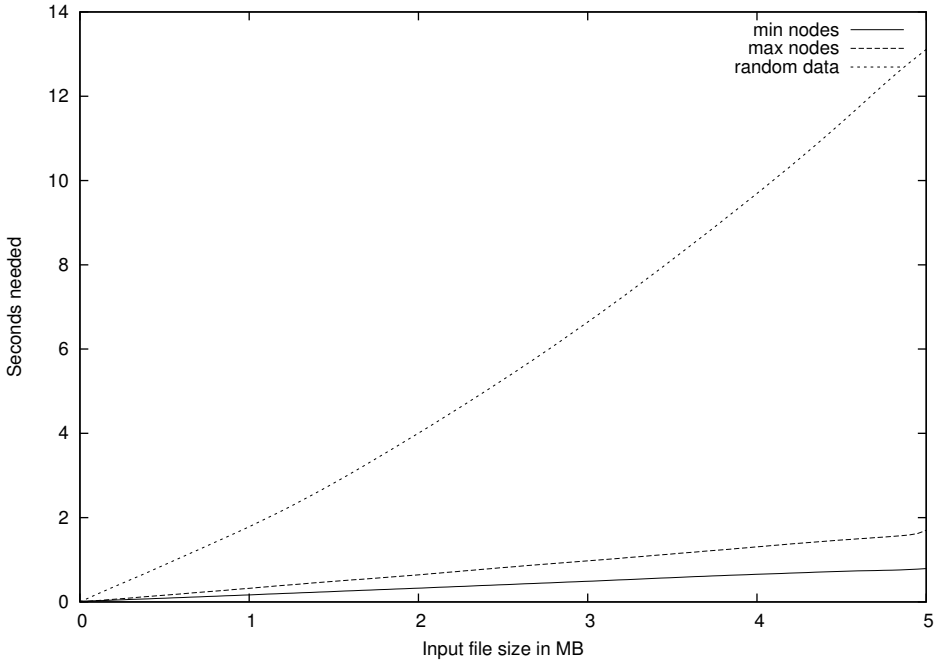

Figure 5.2: The time needed to load a file

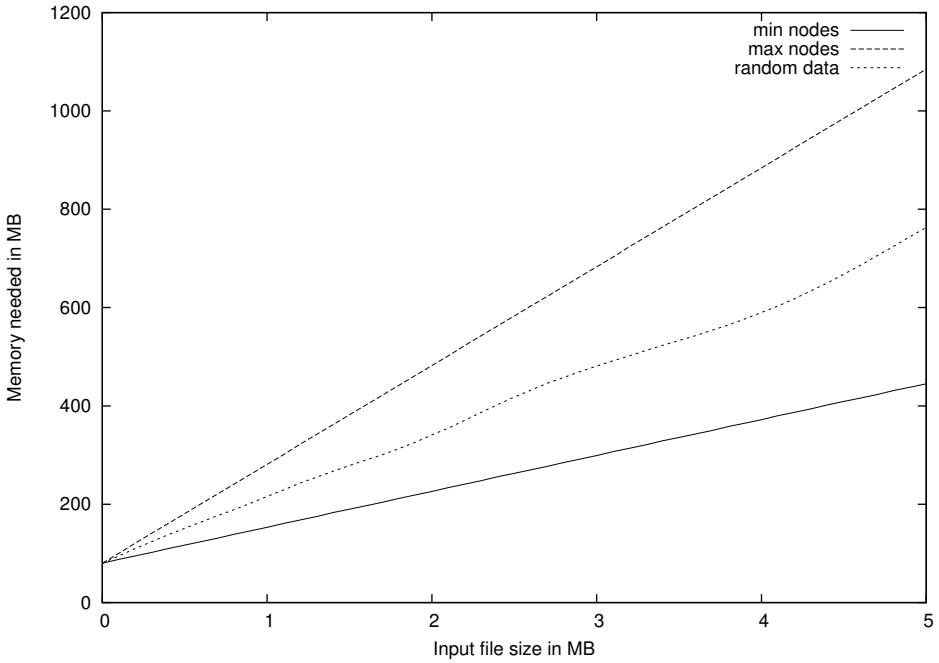

Figure 5.3: The memory needed to load a file

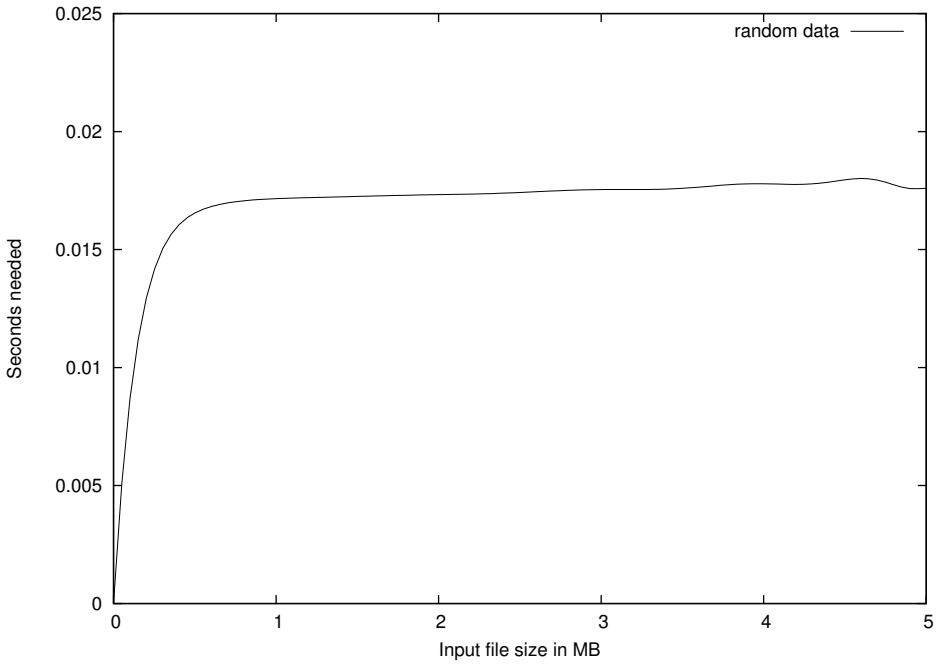

Figure 5.4: The time needed to run a query. For the other two types of input this is 10^{-6} seconds. This graph clearly shows the effects of the cache.

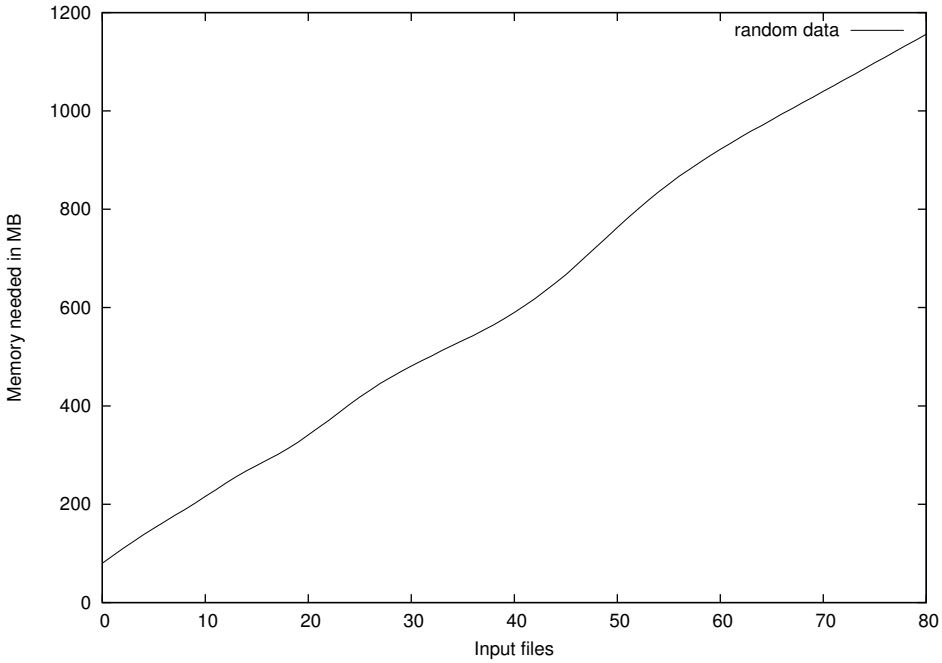

Figure 5.5: The memory needed to load multiple 100 kB random files

5.2 Real world results

5.2.1 The complete works of Shakespeare

This thesis does a duplication analysis on the complete works of Shakespeare. The complete archive was obtained from the website of James Farrow [11].

The complete works consists of 4 900 971 bytes. After loading these, the suffix tree contains 7 331 781 nodes. For this 921 MB memory is used by the duplication detection engine and it took 4.5 seconds to load all files.

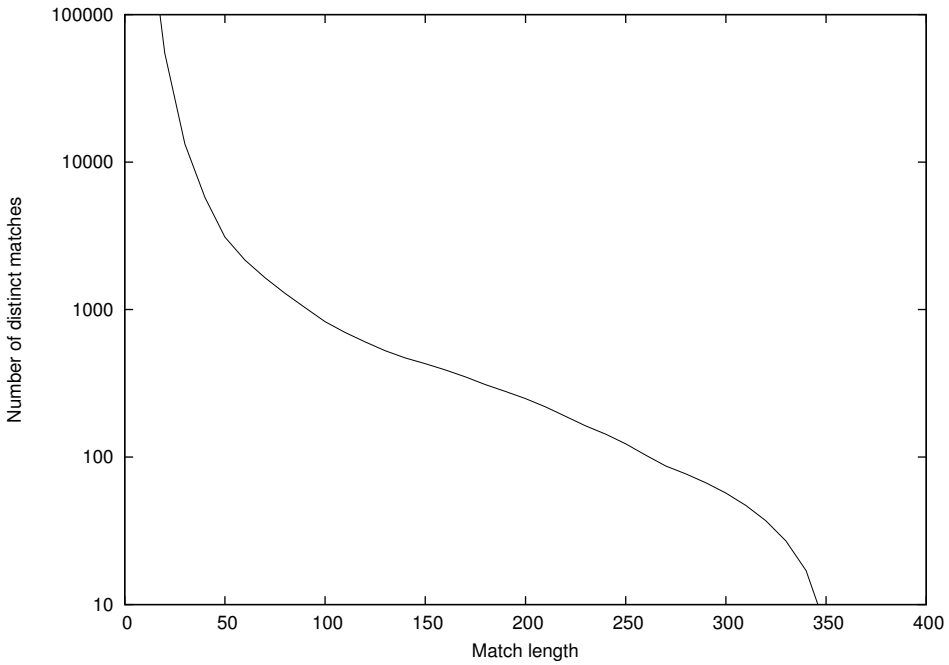

Figure 5.6: Number of distinct matches for different lengths in Shakespeare's all works

Note here that if the documents contain for example a 350 byte long duplication, then when the engine is looking for 200 byte long duplicates this duplicate will be reported 150 times.

5.2.2 The complete works of Charles Dickens

The complete works of Dickens consists of 30 MB. To load all of this DDD needs 5.4 GB of memory. In order to run the analysis on this input a machine with lots of memory was used which is not the same as in the other tests. 28 873 092 characters were loaded into DDD and its suffix tree representation consisted of 43 817 653 nodes. The analysis' first goal was to find the largest fragment found in multiple places. This turns out to be a speech he gave at a public dinner on the 18th of April, 1868. A fragment of this speech was also found in the postscripts of *American Notes: For General Circulation* and *Life and Adventures of Martin Chuzzlewit.*

So much of my voice has lately been heard in the land, that I might have been contented with troubling you no further from my present standing-point, were it not a duty with which I henceforth charge myself, not only here but on every suitable occasion, whatsoever and wheresoever, to express my high and grateful sense of my second reception in America, and to bear my honest testimony to the national generosity and magnanimity. Also, to declare how astounded I have been by the amazing changes I have seen around me on every side, - changes moral, changes physical, changes in the amount of land subdued and peopled, changes in the rise of vast new cities, changes in the growth of older cities almost out of recognition, changes in the graces and amenities of life, changes in the Press, without whose advancement no advancement can take place anywhere. Nor am I, believe me, so arrogant as to suppose that in five and twenty years there have been no changes in me, and that I had nothing to learn and no extreme impressions to correct when I was here first. And this brings me to a point on which I have, ever since I landed in the United States last November, observed a strict silence, though sometimes tempted to break it, but in reference to which I will, with your good leave, take you into my confidence now. Even the Press, being human, may be sometimes mistaken or misinformed, and I rather think that I have in one or two rare instances observed its information to be not strictly accurate with reference to myself. Indeed, I have, now and again, been more surprised by printed news that I have read of myself, than by any printed news that I have ever read in my present state of existence. Thus, the vigour and perseverance with which I have for some months past been collecting materials for, and hammering away at, a new book on America has much astonished me; seeing that all that time my declaration has been perfectly well known to my publishers on both sides of the Atlantic, that no consideration on earth would induce me to write one. But what I have intended, what I have resolved upon (and this is the confidence I seek to place in you) is, on my return to England, in my own person, in my own journal, to bear, for the behoof of my countrymen, such testimony to the gigantic changes in this country as I have hinted at to-night. Also, to record that wherever I have been, in the smallest places equally with the largest, I have been received with unsurpassable politeness, delicacy, sweet temper, hospitality, consideration, and with unsurpassable respect for the privacy daily enforced upon me by the nature of my avocation here and the state of my health. This testimony, so long as I live, and so long as my descendants have any legal right in my books, I shall cause to be republished, as an appendix to every copy of those two books of mine in which I have referred to America. And this I will do and cause to be done, not in mere love and thankfulness, but because I regard it as an act of plain justice and honour.

Figure 5.7: This fragment is a part of a Dickens speech. It can be also found in two other works of his.

Furthermore, the analysis had the goal to find the most used phrases in all works. The tool was set to find the phrase which are at least 30 characters long and occur at least 20 times. The analysis found the following phrases:

- with his hands in his pockets

- the opposite side of the way
- one thousand seven hundred and
- under existing circumstances
- I am very much obliged to you
- one thousand six hundred and

5.2.3 The Holy Bible

The bible consists of single text file obtained from Project Gutenberg [12]. This text file is 4 452 069 bytes long and it takes 3.1 seconds to load it into DDD and the engine uses 860 MB memory to store it (the suffix tree consists 6 906 427 nodes). There was an exact match found in two different books. The following fragment was found in both *The First Book of the Kings, 7:25* and *The Second Book of the Chronicles, 4:4*:

> It stood upon twelve oxen, three looking toward the north, and
> three looking toward the west, and three looking toward the south, and
> three looking toward the east: and the sea was set above upon them,
> and all their hinder parts were inward.

Figure 5.8: A fragment of the Bible which has been found in two distinct books

5.2.4 Linux kernel version 3.3 source code

As first step we look at the sources in the kernel subdirectory. This directory contains the core source files of the kernel and these files represent the highest quality source files in the kernel. By themselves these files occupy 2 361 649 bytes but after the syntax transformer transformation the data is reduced to 438 655 bytes. All of this is loaded under 0.2 seconds and the resulting suffix tree consists of 754 380 nodes which takes up 162 MB of memory.

The most common type of code which is frequently reported is the repetitive code which usually consists of switches, enumerations, or big tables. For example fragment 5.9 matches both 5.10 and 5.11. Another example is shown in 5.12.

```
case AUDIT_COMPARE_UID_TO_OBJ_UID:
        return audit_compare_id(cred->uid,
                                name, offsetof(struct audit_names, uid),
                                f, ctx);
case AUDIT_COMPARE_GID_TO_OBJ_GID:
        return audit_compare_id(cred->gid,
                                name, offsetof(struct audit_names, gid),
                                f, ctx);
case AUDIT_COMPARE_EUID_TO_OBJ_UID:
        return audit_compare_id(cred->euid,
                                name, offsetof(struct audit_names, uid),
                                f, ctx);
case AUDIT_COMPARE_EGID_TO_OBJ_GID:
        return audit_compare_id(cred->egid,
                                name, offsetof(struct audit_names, gid),
                                f, ctx);
```

Figure 5.9: A fragment from auditsc.c

```
case AUDIT_COMPARE_EUID_TO_OBJ_UID:
        return audit_compare_id(cred->euid,
                                name, offsetof(struct audit_names, uid),
                                f, ctx);
case AUDIT_COMPARE_EGID_TO_OBJ_GID:
        return audit_compare_id(cred->egid,
                                name, offsetof(struct audit_names, gid),
                                f, ctx);
case AUDIT_COMPARE_AUID_TO_OBJ_UID:
        return audit_compare_id(tsk->loginuid,
                                name, offsetof(struct audit_names, uid),
                                f, ctx);
case AUDIT_COMPARE_SUID_TO_OBJ_UID:
        return audit_compare_id(cred->suid,
                                name, offsetof(struct audit_names, uid),
                                f, ctx);
```

Figure 5.10: Another fragment from auditsc.c which matches figure 5.9

```
case AUDIT_COMPARE_EGID_TO_OBJ_GID:
        return audit_compare_id(cred->egid,
                                name, offsetof(struct audit_names, gid),
                                f, ctx);
case AUDIT_COMPARE_AUID_TO_OBJ_UID:
        return audit_compare_id(tsk->loginuid,
                                name, offsetof(struct audit_names, uid),
                                f, ctx);
case AUDIT_COMPARE_SUID_TO_OBJ_UID:
        return audit_compare_id(cred->suid,
                                name, offsetof(struct audit_names, uid),
                                f, ctx);
case AUDIT_COMPARE_SGID_TO_OBJ_GID:
        return audit_compare_id(cred->sgid,
                                name, offsetof(struct audit_names, gid),
                                f, ctx);
```

Figure 5.11: Yet nother fragment from auditsc.c which matches figure 5.9

```
extern const struct kernel_symbol __start___ksymtab[];
extern const struct kernel_symbol __stop___ksymtab[];
extern const struct kernel_symbol __start___ksymtab_gpl[];
extern const struct kernel_symbol __stop___ksymtab_gpl[];
extern const struct kernel_symbol __start___ksymtab_gpl_future[];
extern const struct kernel_symbol __stop___ksymtab_gpl_future[];
```

Figure 5.12: This is a fragment from module.c. As each line looks the same to DDD this can generate multiple matches depending on the minimum match size in similar matter to figures 5.9, 5.10 and 5.11.

5.2.5 rxvt-unicode terminal emulator version 9.14 source code

rxvt-unicode is a well known terminal emulator for Linux written in C++ [13]. Its source code consists of 749 134 bytes. After running through the syntax transformer this was reduced to 138 825 bytes.

Although switches and tables generated lots of false positives, there were found some real duplications which could be improved or abstracted out:

```
/* fill our lookup tables */
for (i = 0; i <= mask_r>>sh_r; i++)
  {
    uint32_t tmp;
    tmp = i * high.r;
    tmp += (mask_r>>sh_r) * low.r;
    lookup_r[i] = (tmp/65535)<<sh_r;
  }
for (i = 0; i <= mask_g>>sh_g; i++)
  {
    uint32_t tmp;
    tmp = i * high.g;
    tmp += (mask_g>>sh_g) * low.g;
    lookup_g[i] = (tmp/65535)<<sh_g;
  }
for (i = 0; i <= mask_b>>sh_b; i++)
  {
    uint32_t tmp;
    tmp = i * high.b;
    tmp += (mask_b>>sh_b) * low.b;
    lookup_b[i] = (tmp/65535)<<sh_b;
  }
```

Figure 5.13: This is a fragment from background.C. The three loops contain the almost same code. This could be abstracted into a separate function.

Chapter 6

Conclusion

6.1 Summary

This thesis gave an introduction to the field of duplication searching and an overview of the related work.

A generic tool has been implemented for finding duplicates in various types of documents. The tool took the approach of modularized components and consists of three main components. The heart of the tool is an implementation of the rather complicated suffix tree algorithm. The implementation of this algorithm was done through the careful selection of the data structures and through the careful operations on top of these data structures. The tool also included a set of input transformer modules for transforming various sorts of inputs like text files and source files. The tool also included an output presenter which transformed the results of the tool into a human readable HTML files.

The algorithm's running time and memory characteristics were measured both on synthetic inputs and on real world inputs like the complete works of Shakespeare and Charles Dickens or projects like the Linux kernel source and the rxvt-unicode terminal emulator. The results show that the tool presented is very handy for analyzing the source code for searching refactoring and abstraction possibilities.

6.2 Further development possibilities

The tool described in this thesis can be further developed in various ways. Adding more input transformers is one such option. For example an input transformer could strip the markup from HTML and LaTeX files. Another input transformer could strip the headers and then reduce the quality for images in order to seek duplicates in a set of images.

The tool's presentation facilities can also be extended by various GUI tools for

presenting the results and running the queries interactively via the GUI.

Improvements can also be made to the main engine of the tool. One such improvement idea is to add an option to rebuild the whole tree to generate a more cache friendly tree because the results showed that cache-friendliness gives a big boost in performance.

Bibliography

[1] B. S. Baker: *On finding duplication and near-duplication in large software systems*, WCRE '95 Proceedings of the Second Working Conference on Reverse Engineering

[2] Cory Kapser, Michael W. Godfrey: *"Cloning Considered Harmful" Considered Harmful*, WCRE '06 Proceedings of the 13th Working Conference on Reverse Engineering

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

[4] Ira D. Baxter, et al.: *Clone Detection Using Abstract Syntax Trees*, ICSM '98 Proceedings of the International Conference on Software Maintenance

[5] Matthias Rieger, Stéphane Ducasse: *Visual Detection of Duplicated Code*, ECOOP '98 Proceedings of the Workshop ion on Object-Oriented Technology

[6] Stéphane Ducasse, Matthias Rieger, Serge Demeyer: *A Language Independent Approach for Detecting Duplicated Code*, ICSM '99 Proceedings of the IEEE International Conference on Software Maintenance

[7] Hoad, Timothy; Zobel, Justin:, *Methods for Identifying Versioned and Plagiarised Documents*, Journal of the American Society for Information Science and Technology 54 (3): 203–215

[8] Si, Antonio; Leong, Hong Va; Lau, Rynson W. H.: *"CHECK: A Document Plagiarism Detection System"*, SAC '97: Proceedings of the 1997 ACM symposium on Applied computing, ACM, pp. 70–77

[9] E. Ukkonen: *Constructing suffix trees on-line in linear time*, Proc. Information Processing 92, Vol. 1, IFIP Transactions A-12, 484-492, Elsevier 1992

[10] Valgrind, *http://valgrind.org*, May 01, 2012.

[11] The Collected Works of Shakespeare *http://sydney.edu.au/engineering/it/ matty/Shakespeare/*, May 01, 2012

[12] The Bible, *http://www.gutenberg.org/ebooks/10*, May 01, 2012.

[13] rxvt-unicode, *http://software.schmorp.de/pkg/rxvt-unicode.html*, May 01, 2012.

Appendix A

CD-ROM contents

The source files and documentation can be found in the following directories:

- source code in the `app/` folder
- this paper is in the `docs/` folder