UNIVERSITY
OF OSLO

Master's thesis

CC-LSP: Language and editor agnostic code clone detection

Incremental code clone detection for text editors

Jakob Konrad Hansen

60 ECTS study points

Department of Informatics
Faculty of Mathematics and Natural Sciences

Spring 2023

Jakob Konrad Hansen

CC-LSP: Language and editor agnostic code clone detection

Incremental code clone detection for text editors

Abstract

Duplicated code is bad

Contents

Chapter 1

Introduction

Duplicate code is bad

1.1 Motivation

Many algorithms exist for code clone detection, but not many are incremental. Finding incremental algorithms is interesting for use-cases such as editors and git revisions of the same project...

Chapter 2

Background

2.1 Introduction

Refactoring is the process of restructuring source code in order to improve the internal behavior of the code, without changing the external behavior [7, p. 9]. Refactoring on source code is often performed in order to eliminate instances of bad design quality in code, otherwise known as code smells.

A study conducted by Diego Cedrim et al. has shown that while developers tend to refactor smelly code, they are rarely successful at eliminating the smells they are targeting [5]. They also discovered that a large portion of refactorings tend to make the code quality worse. Automated tools which help developers make better refactorings and perform code analysis could be a solution to this problem.

Duplicated code is a code smell which occurs in practically every large software project. Code clone analysis has recently become a highly active field of research and many tools have been developed to detect duplicated code [11, p. 7]. However, few of these have the capability of detecting intricate types of duplicated code and managing them in a real-time IDE environment.

This thesis will present a tool and possibly techniques for clone detection and management in real-time, which runs in modern IDE environments. The thesis will explore the topics such as finding and managing clones in real-time, incremental analysis of source code and providing clone management and refactoring tools in an editor and language agnostic environment.

2.2 Background

2.2.1 Software quality and duplicated code

Software quality is hard to define. The term "quality" is ambiguous and is in the case of software quality, multidimensional. Quality in itself has been defined as "conformance to requirements" [6, p. 8]. In software, A simple measure of "conformance to requirements" is a lack of bugs. However, software quality is often measured in other metrics, including metrics which are not directly related to functionality [12, p. 29]. These metrics often include maintainability, analyzability and changeability.

These metrics are affected negatively by duplicated code, code which is more or less copied to different locations in the source code. Multiple studies have shown that software projects typically contains $10 - 15\%$ duplicated code [1]. Therefore, research into tools and techniques which can reduce duplicated code will be of benefit to almost all software.

As stated, duplicated code damages software quality in software projects. Duplicated code can lead to a plethora of antipatterns, and will often lead to an increase in technical debt. Technical debt occurs when developers make technical compromises that are expedient in the short term, but increases complexity in the long-term [2, p. 111]. An example of this in the context of duplicated code is the "Shotgun-Surgery" [7, p. 66] antipattern. This antipattern occurs when a developer wants to implement a change, but needs to change code at multiple locations for the change to take effect. This is a typical situation which slows down development and reduces maintainability when the amount of duplicated code increases in a software project.

2.2.2 Code clones

Duplicated code is often described as "code clones".

Definition 1 (Code snippet). *A code snippet is a piece of contiguous source code in a larger software system.*

Definition 2 (Code clone). *A code clone is a code snippet which is equal to or similar to another code snippet. The two code snippets are both code clones, and together they form a code clone pair. Similarity is determined by some metric such as number of equal lines of code.*

Definition 3 (Clone set). *A clone set is a set of code snippets where all snippets are considered clones of each other.*

Clone relation

The clone relation is a relation between code snippets which defines pairs of clones. The clone relation is reflexive and symmetric, but not always transitive. The transitive property depends on the threshold for similarity when identifying code clones. Given

$$a \xleftrightarrow{clone} b \xleftrightarrow{clone} c$$

where a, b, c are code snippets and $\xleftrightarrow{clone}$ gives the clone relation, a is a clone of b, but not necessarily similar enough to be a clone of c, depending on the threshold for similarity.

If the threshold for similarity is defined such that only equal clones are considered clones, the relation becomes transitive, and equivalence classes form clone sets.

Code clone types

Code clones are generally classified into four types [11]. The types classify code snippets as code clones with an increasing amount of leniency. Therefore, Type-1 code clones are very similar, while Type-4 clones are not necessarily similar at all. When defining types, it is the syntactic and structural differences which is compared, not functionality. The set of code clones classified by a code clone type is also a subset of the next type, meaning all Type-1 clones are also Type-2 clones, but not vice versa.

The code clone types are defined as follows:

Type-1 clones are syntactically identical. The only differences allowed are elements without meaning, like comments and white-space. Example:

```
// Clone 1
for (int i = 0; i < 10;    i++) {
    print(i);
}
```

```
// Clone 2
for (int i = 0; i < 10; i++) {
    // A comment without meaning
    print(i);
}
```

Type-2 clones are structurally identical. Possible differences include identifiers, literals and types. Type-2 clones are relatively easy to detect by consistently renaming identifiers and literals [26, p. 2]. This type of clone is relevant to consider in merging scenarios because this type of clone is relatively simple to parameterize in order to merge two Type-2 clones. Example:

```
// Clone 1
for (int i = 0; i < 10; i++) {
    print(i);
}
```

```
// Clone 2
for (int (*\textbf j*) = (*\textbf 1*); (*\textbf
    print((*\textbf j*));
}
```

Type-3 clones are required to be structurally similar, but not equal. Differences include statements which are added, removed or modified. This clone type relies on a threshold θ which determines how structurally different snippets can be to be considered Type-3 clones [11]. The granularity for this difference could be based on differing tokens, lines, etc. Detecting this type of clone is hard. Example:

```
// Clone 1
for (int i = 0; i < 10; i++) {
    print(i);
}
```

```
// Clone 2
for (int i = 0; i < 10; i++) {
    print(i);
    (*\textbf{int x = 10;}*)
}
```

In this example there is a one line difference between the two snippets, so if $\theta \geq 1$, the two snippets would be considered Type-3 clones.

Type-4 clones have no requirement for syntactical or structural similarity. Therefore, the only requirement is equal functionality. Detecting this type of clone is very challenging, but attempts have been made using program dependency graphs [25]. The following example shows two code snippets which have no clear syntactic or structural similarity, but is functionally equal:

```
// Clone 1
print((n*(n-1))/2)

print(sum);
```

```
// Clone 2
int sum = 0;
for (int i = 0; i < n; i++) {
    for (int j = i+1; j < n; j++) {
        sum++;
    }
}
print(sum);
```

Type-1 clones are often referred to as "exact" clones, while Type-2 and Type-3 clones are referred to as "near-miss" clones [26, p. 1].

Code clone detection process and techniques

The Code clone detection process is generally split into (but is not limited to) a set of steps to identify clones [11]. This process is often a pipeline of input-processing steps before finally comparing fragments against each other and filtering. The steps are generally as follows:

1. **Pre-processing**: Filter uninteresting code that we do not want to check for clones, for example generated code. Then partition code into a set of fragments, depending on granularity such as methods, files or lines.
2. **Transformation**: Transform fragments into an intermediate representation, with a source-map back to the original code.
   (a) Extraction: Transform source code into the input for the comparison algorithm. Can be tokens, AST, dependency graphs, suffix tree, etc.
   (b) Normalization: Optional step which removes superficial differences such as comments, whitespace and identifier names. Often useful for identifying type-2 clones.
3. **Match detection**: Perform comparisons which outputs a set of candidate clone pairs.

4. **Formatting**: Convert candidate clone pairs from the transformed code back to clone pairs in the original source code.
5. **Post-processing/Filtering**: Ranking and filtering manually or with automated heuristics
6. **Aggregation**: Optionally aggregating sets of clone pairs into clone sets

Not all clone detection techniques will necessarily follow all these steps.

Matching techniques are techniques which can be applied to match source-code to detect clone-pairs. The matching technique will also require specific pre-processing to be done in the earlier steps, for example creating an AST. Some of the most explored techniques are as follows [20]:

Text-based approaches do very little processing on the source code before comparing. Simple techniques such as fingerprinting or incremental hashing have been used in this approach. Dot plots have also been used in newer text-based approaches, placing the hashes of fragments in a dot plot for use in comparisons.

Token-based approaches transform source code into a stream of tokens, similar to lexical scanning in compilers. The token stream is then scanned for duplicated subsequences of tokens. Since token streams can easily filter out superficial differences such as whitespace, indentation and comments, this approach is more robust to such differences. Concrete names of identifiers and values can be abstracted away when comparing the token-stream, therefore Type-2 clones can easily be identified. Type-3 clones can also be identified by comparing the fragments tokens and keeping clone pairs with a lexical difference lower than a given threshold. This can be solved with dynamic programming [3]. A common approach to detect clones using token-streams is with a suffix-tree. A suffix-tree can solve the *Find all maximal repeats* problem efficiently, which in essence is the same problem as finding clone pairs. This algorithm can also be improved to use a suffix-array instead, which requires less memory.

Syntactic approaches transform source code into either concrete syntax trees or abstract syntax trees and find clones using either tree matching algorithms or structural metrics. For tree matching, the common approach is to find similar subtrees, which are then deemed as clone pairs. One way of finding similar subtrees is to hash subtrees into buckets and compare them with a tolerant tree matching algorithm. Variable names, literal values and other source may be abstracted to find Type-2 clones more easily. Metrics-based techniques gather metrics for code fragments in the tree and uses the metrics to determine if the fragments are clones or not. One way is to use fingerprint functions where the fingerprint includes certain metrics, and compare the fingerprints of all fragments to find clones.

Chunk-based approaches decompose chunks of source code into signatures which are compared. Chunk-size is based on selected granularity, which can be functions, blocks, etc. Signatures can for example be based on some software metrics. Machine learning has been used in this approach using methods as chunks and token-frequency within the method as signature. A Deep Neural Network trained on this data can then be used to classify two chunks as clone or non-clone [15].

Hybrid approaches combine multiple approaches in order to improve detection. For example Zibran et al. developed a hybrid algorithm combining both token-based suffix trees for Type-1 and Type-2 clone detection, with a k-difference dynamic programming algorithm for Type-3 clone detection [26].

2.2.3 Incremental editing and analysis

While writing code, programmers usually only edit small portions of text at a time. One "edit" will therefore only affect small parts of the internal representations of the code which most tools use to perform analysis. Reusing parts of this representation would therefore be faster and allow programming tools to scale better.

Incremental parsing

Incremental parsing is the process of reparsing only parts of a syntax tree whenever an edit is performed. The motivation behind incremental parsing is to have a readily available syntax tree after every edit, while doing as little computing as possible to build it.

Ghezzi and Mandrioli introduced in 1979 the notion of incremental parsing, and introduced an incremental parser for LR $\wedge$ RL grammars. However, they were aware that this algorithm was both slow and did not allow expressive enough grammars. [8]

Tim A. Wagner et al. [24] later published a large work on incremental software development environments, presenting many novel algorithms and techniques for incremental tooling in programming environments.

"Tree-sitter" is a parser generator tool which specializes in incremental parsing. Inspired by Wagner's work, it supports incremental parsing, error recovery and querying for specific nodes and subtrees [4]. These features combined allow Tree-sitter to become a powerful tool for analysis and has been used for editor features such as syntax-highlighting, refactoring and code navigation.

Incremental clone detection

In order to incrementally detect code clones, an algorithm which first calculates the initial code clones is run, and for successive revisions of the source code, this list is incrementally updated, more efficiently than the initial run. Different approaches have been used to accomplish this.

Göde and Koschke [9] proposed the first incremental clone detection algorithm. The algorithm employs a generalized suffix tree in which the amount of work of updating is only dependent on the size of the edited code. This approach is limited in scalability, as generalized suffix trees require a substantial amount of memory.

Nguyen et al. [16] showed that an AST-based approach utilizing "Locality-Sensitive Hashing" can detect clones incrementally with high precision, and showed that incremental updates could be done in real-time ($<$ 1 second) for source code with a size of 300 KLOC.

Hummel et al. [10] later introduced the first incremental, scalable and distributed clone detection technique for Type-1 and Type-2 clones. This approach utilizes a custom "clone index" data structure which can be updated efficiently. The implementation of this data structure is similar to that of an inverted index.

More recently, Ragkhitwetsagul and Krinke [18] presented the tool "Siamese", which uses a novel approach of having multiple intermediate representations of source code to detect a high number of clones with support for incremental detection. The tool can detect up to Type-3 clones, but will only give clones based on "queries" given to it by the user. Queries are either files or methods in source-code, which are then checked for existing code clone.

2.2.4 IDE tooling

Existing IDE tools for clone management

Developers are not always aware of the creation of clones in their code. Clone aware development means having clone management as a part of the software development process. Since code clones can be hard to keep track of and manage, tools which help developers deal with clones are useful. However, Mathias Rieger et al. claims that a problem with many detection tools is that the tools "report large amounts data that must be treated with little tool support" [19, p. 1]. Detecting and eliminating clones early in their lifecycle with IDE integrated tools could be a solution to the problem of dealing with too many clones.

There are many existing clone management tools, and the following section will go over tools which are integrated into an IDE and offer services to the programmer while developing in real-time.

The IDE-based tools which exist can be categorized as follows [22, p. 8]:

- *Copy-paste-clones:* This category of tools deals only with code snippets which are copy-pasted from another location in code. These tools therefore only track clones which are created when copy-pasting, and does not use any other detection techniques. Therefore, this type of tool is not suitable for detecting clones which are made accidentally, since developers are aware that they are creating clones when pasting already existing code snippets.
- *Clone detection and visualization tools:* This category of tools has more sophisticated clone detection capabilities and will detect code clones which occur accidentally.
- *Versatile clone management:* This category of tools covers tools which provide more services than the above. Services like refactoring and simultaneous editing of clones fall under this category.

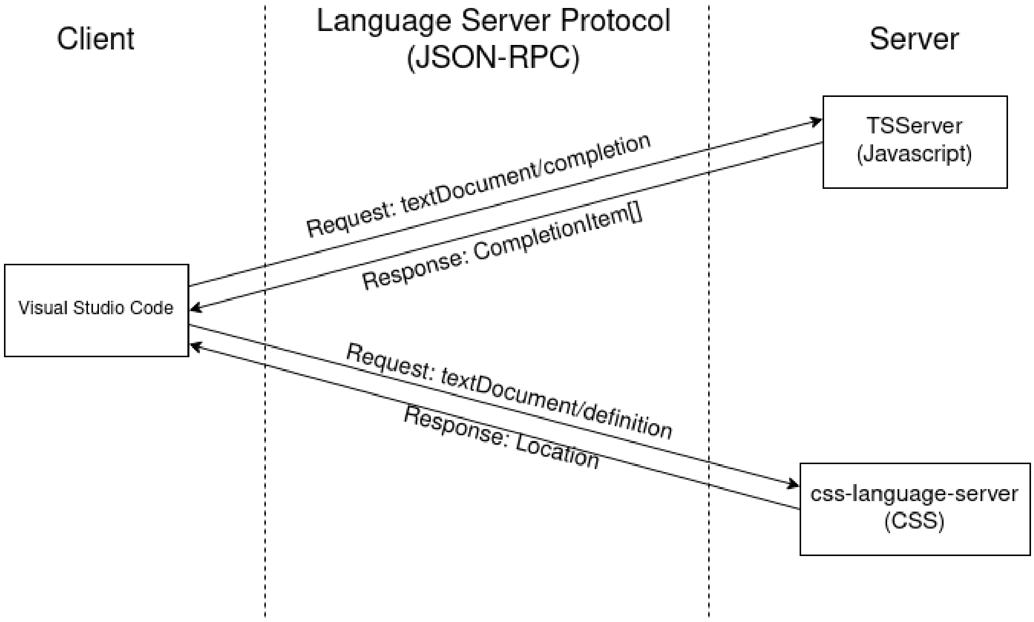

Figure 2.1: Example client-server interaction using LSP

There are a few existing IDE-tools which have seen success in real-time detection of clones:

- Minhaz et al. introduced a hybrid technique for performing real-time focused searches, i.e. searching for code clones of a selected code snippet. This technique can also detect Type-3 clones [26]. It was later used in the tool *SimEclipse* [22] which is a plugin for the Eclipse editor. Since this tool can only detect clones of a code snippet which the developer actively selects, this tool is not well suited for finding accidental clones and tracking clones in areas.
- Another tool, SHINOBI, which is a plugin for the Visual Studio editor, can detect code clones in real-time without the need of the developer to select a code snippet. It can detect Type-1 and Type-2 code clones and uses a token-based suffix array index approach to detect clones incrementally [14].
- The modern IDE IntelliJ has a built-in duplication detection and refactoring service, it is able to detect Type-1 and (some) type-2 code clones at a method granularity and refactors by replacing one of the clones with a method call to the other.

The Language Server Protocol

Usually, static analysis tools which integrate with IDEs are tightly coupled to a specific IDE and its APIs, like parsing and refactoring support. This makes the tools hard to utilize a tool in another IDE, since the API's the tool utilizes is no longer available. In order to make IDE-based static analysis tools more widely available, it would be interesting to determine if such tools could be made editor agnostic.

The Language Server protocol (LSP) is a protocol which specifies interaction between a client (IDE) and server in order to provide language tooling for the client. The goal of the protocol is to avoid multiple implementations of the same language tools for every IDE and every language, allowing for editor agnostic tooling. Servers which implement LSP will be able to offer IDEs code-completion, error-messages, go-to-definition and more. LSP also specifies generic code-actions and commands, which the LSP server provides to the client in order to perform custom actions defined by the server.

Figure 2.1 shows a sample interaction between client and server using LSP. The client sends requests to a server in the form of JSON-RPC messages, and the server sends a corresponding response, also in the form of JSON-RPC messages.

2.3 Preliminary algorithms and data structures

The following algorithms and data structures will be useful in following chapters to define the detection algorithm

2.3.1 Suffix trees

A classic algorithm for code clone detection traverses a suffix-tree in order to find maximal repeats in all suffixes of the input string T.

The suffix tree of a string T is a compressed trie where all the suffixes of T have been inserted. The tree is compressed by combining consecutive nodes in a row which has only one child into a single node.

Suffix trees can be constructed in linear time with Ukkonen's algorithm which builds a larger and larger suffix tree by inserting characters one by one and utilizing some tricks to avoid inserting suffixes before it needs to, to lower the complexity. [23]

This data structure facilitates solving the maximal repeat problem. A repeat in a string T is a substring that occurs at least twice in T. A maximal repeat in T is a repeat which is not a substring of another repeat in T, meaning that the maximal repeat cannot be extended in any direction to form a bigger repeat. This problem can be solved with a suffix tree by the following theorem:

Theorem 1 (Repeats in suffix tree). *Every internal node in a suffix tree corresponds to a substring which is repeated at least twice in T. The substring is found by concatenating the strings found on the path from the root of tree to the internal node.*

This theorem is explained by the fact that any internal node has at least two children, and a node having two children means that two suffixes share the same prefix up to that point. An algorithm which finds the longest maximal repeat would find the internal node with represents the longest string.

The classic algorithm[26, 9] in terms of finding duplication in a string (such as source code) using suffix trees would find all repeats of length k where k is the threshold for how long a clone

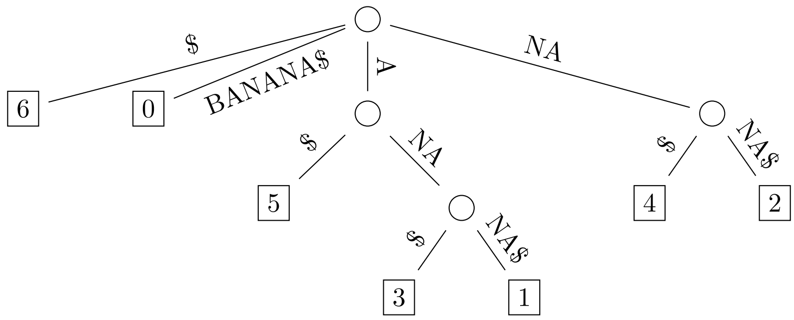

Figure 2.2: Suffix tree for T=BANANA$

needs to be. This can be found by traversing the suffix tree and looking at all internal nodes which represent a string of length $\geq k$. Every internal node which represents a string which is $\geq k$ would correspond to a substring of the source code which occurs at least twice. Finding where the duplication occurs can be done by finding all the leaves of the internal node, which holds the position where the suffix starts in T. Since a substring can have multiple repeats of different lengths longer than k, different strategies can be used to select which substrings are selected or not, such as filtering out repeats which are not maximal or repeats which overlap each other.

2.3.2 Suffix arrays

The suffix array (SA) of a string T contains a lexicographical sorting of all suffixes in T. The suffix array does not contain the actual suffixes, but it contains integers pointing to the index where the suffix starts in T. Conversely, the inverse suffix array (`ISA`) contains integers describing which rank a suffix has. `ISA` is therefore the inverse array of `SA`, such that if `SA[i] = n`, then `ISA[n] = i`.

Definition 4 (Suffix array)**.** *Let T be a text of length N. The suffix array* `SA` *of T is an array of length N where* `SA[i] = n` *if the suffix at* `T[n..N-1]` *is the ith smallest suffix in T lexicographically.*

Definition 5 (Inverse suffix array)**.** *Let T be a text of length N. The inverse suffix array ISA of T is an array of length N where* `ISA[i] = n` *if the suffix at* `T[i..N-1]` *is the nth smallest suffix in T lexicographically.*

The Longest-common prefix (LCP) array of a string T of length N is an array of length N such that each element contains the length of the common prefix between two suffixes in T. The suffixes are ordered in the same order as the suffix array. Since the suffix array represents suffixes in a sorted order, the prefix length between adjacent suffixes in SA will be the longest possible common prefix for each suffix. These values are the values in the LCP array.

Definition 6 (LCP array)**.** *Let T be a text of length N and SA be the suffix array of T. The LCP array of T is an array of length N where* `LCP[i] = n` *if the suffix* `T[SA[i]..N]` *and* `T[SA[i-1]..N]` *has a common prefix of length* `n`*.* `LCP[0]` *is undefined or 0.*

Index	Suffix
0	BANANA$
1	ANANA$
2	NANA$
3	ANA$
4	NA$
5	A$
6	$

(a) Suffixes

Index	Suffix
6	$
5	A$
3	ANA$
1	ANANA$
0	BANANA$
4	NA$
2	NANA$

(b) Sorted suffixes

Index	SA	ISA	LCP
0	6	4	-1
1	5	3	0
2	3	6	0
3	1	2	0
4	0	5	0
5	4	1	0
6	2	0	0

(c) SA, ISA and LCP

Table 2.1: $T = \text{BANANA\$}$

Table 2.1 shows the suffix array of a text T = BANANA$ and the correlation between the sorted suffixes and SA, ISA and LCP.

Suffix arrays can be constructed in linear time in terms of the length of T. Many so-called suffix array construction algorithms (SACA) have been discovered in the last decade[13], many of which run in linear-time. An algorithm which has been shown to be very efficient in practice is Nong and Chan's[17] algorithm based on induction sorting. Since ISA is the inverse of the suffix array, it can also be constructed in linear time by first constructing the suffix array.

The LCP array can also be constructed in linear time, utilizing SA and ISA.

2.3.3 Dynamic bitsets

A bitset is an array of bits, each bit representing either the value true or false. A bit with the value of 1 is usually referred to as a set bit, and a value of 0 is referred to as an unset or cleared bit. Bitsets have at least operations for setting the value at a position, and looking up the value at a position. Bitsets are useful for many problems, especially as a "succinct data structure". A succinct data structure is a data structure which attempts to use an amount of memory close to the theoretic lower bound, while still allowing effective queries on it. For example for a string of length N, we could store up to $O(n \log \sigma)$ bits before the bit vector exceeds the size of the string itself, where σ is the size of the string's alphabet.

The most well known query to do on bitsets is the rank/select queries.

Definition 7 (Rank query)**.** *A rank query $rank_1(i)$ on a bitset B, returns the number of set bits up to, but not including position i. Conversely, $rank_0(B)$ returns the number of unset bits up to i.*

Definition 8 (Select query)**.** *A select query $select_1(i)$ on a bitset B, returns the position of the ith set bit in B. Conversely, $select_0(i)$ returns the position of the ith unset bit in B.*

Jacobson's rank can calculate rank and select on static bitsets in $O(1)$ time by pre-calculating all answers in a space efficient table.

Definition 9 (Dynamic bitset)**.** *A dynamic bitset is a bitset which in addition to other operations allow inserting and deleting bits (indel operations).*

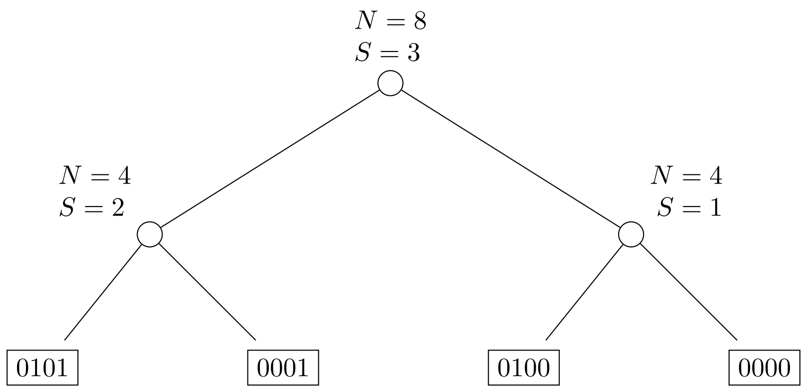

Figure 2.3: Dynamic bitset

An insert operation insert(i, v) on a bitset B inserts the value v at position i in B, pushing all values at position $\geq i$ one position up.

A delete operation delete(i) on a bitset B removes the value at position i in B, pushing all values at position $> i$ one position down.

The standard implementation of a dynamic bitset would implement the whole bitset as a single array of bytes B, which allows for accessing values in $O(1)$ time, but inserting and deleting takes $O(n)$ time, where n is the number of bits in B.

One way to speed up the indel operations would be to represent the bitset as a balanced tree containing multiple smaller bitsets. To represent a bitset of N bits, we can divide the bits into smaller bitsets, such that $O(\frac{n}{\log(n)})$ bitsets each contains $O(log(n))$ bits. Since the bitsets are now of size $O(log(n))$, inserting and deleting takes only $O(log(n))$ time. The bitsets reside at the leaves of our balanced tree, and internal nodes contain only two integers, storing the number of bits in the left subtree (N), and the number of set bits in the left subtree (S). To access, insert or delete on index i, the tree is traversed to find the correct bitset where the ith bit is located, where the operation is done at that position. Finding the correct bitset and position is done by utilizing N and S in each node which is traversed. All operations now run in $O(\log(n))$ time, since traversing the tree to the correct bitset takes $O(\log(\frac{n}{\log(n)}))$ and performing the operation takes $O(\log(n))$ time.

Figure 2.3 shows how a dynamic bitset tree is structured, and Algorithm 1 shows how to access a value in the tree. Traversing the tree to calculate rank and select queries can be done similarly by summing set bits in left-subtrees (rank) or selecting which subtree to descend based on the number of set bits (select).

```
Algorithm access(node, i)
    if isLeaf(node) then
        return node.bitset[i]
    end
    if node.N ≤ i then
        return access(node.left, i)
    end
    return access(node.right, i - node.N)
```

Algorithm 1: Accessing a value in a dynamic bitset

2.3.4 Wavelet trees and wavelet matrices

2.3.5 Burrows-Wheeler transform

The Burrows-Wheeler transform (BWT) is a transform on strings, often done to improve compression. The transform is computed by sorting all "cyclic-shifts" of the string lexicographically and extracting a new string from the last column of the cyclic-shift" matrix. The terminating character $ is always the smallest character lexicographically. Table 2.2 shows the BWT for the string T = BANANA$.

There is a strict correlation between the suffix array of T and the BWT of T. Figure 2.2 also shows that the indices of the sorted cyclic-shifts correspond to exactly the SA of T. This coincides because when sorting cyclic-shifts, everything that occurs after the $ of the cyclic-shift will not affect its lexicographical ordering. This is because no cyclic-shift will have a $ in the same position, so comparing two cyclic shifts lexicographically will always terminate whenever a $ is found. This means that the ordering of cyclic shifts is essentially the same as sorting all suffixes of T. Algorithm 2 shows how the BWT of T can be calculated directly from the SA of T in linear time. Since there is a 1:1 correlation between the SA and BWT, dynamic updates to a BWT would correspond to similar dynamic updates in the SA (and ISA). This will be useful in the following code clone detection algorithm.

```
Algorithm BWT(T, SA)
    N ← len(T)
    bwt ← string of length N
    for i ∈ 0..N do
        pos ← (SA[i] - 1) % N
        bwt[i] = T[pos]
    end
    return bwt
```

Algorithm 2: Calculating the BWT of a string T from its suffix array

An essential property of the BWT is that the transformation is reversible. By examining the BWT of a string T, we see that the BWT is a permutation of T. We will also see that there is a correlation between the characters in the first column of the cyclic-shift matrix and the last column (the BWT).

Index	Cyclic-shift
0	BANANA$
1	ANANA$B
2	NANA$BA
3	ANA$BAN
4	NA$BANA
5	A$BANAN
6	$BANANA

(a) Cyclic shifts

Index	Cyclic-shift
6	$BANAN**A**
5	A$BANA**N**
3	ANA$BA**N**
1	ANANA$**B**
0	BANANA**$**
4	NA$BAN**A**
2	NANA$B**A**

(b) Sorted cyclic shifts and BWT

Table 2.2: T = BANANA$, BWT = ANNB$AA

T can be calculated from the BWT, as long as there is a unique terminating character ($) or the position of the final character is stored. T is computed backwards by starting at the final character ($T[n-1]$) and then finding the cyclic-shift where that character occurs in the first column. The character in the last column of that cyclic-shift is $T[n-2]$. If there are multiple of the same character the nth character of a certain symbol in the last column will map to the nth character in the first column. This process is repeated until we finish the cycle, returning to the final character in the last column. Essentially, this process consists of traversing cyclic-shifts backwards and looking at the final character, which will give us T, since the final character of the cyclic-shift is continually shifting one position.

Determining the previous cyclic-shift requires some insight about the first column of the BWT matrix. It also means that the first column of the BWT matrix contain all the characters in T in a sorted order. Therefore, we can calculate the next cyclic shift from only the last column by determining how many characters are lexicographically smaller than the current character, and also the rank of the current character at this position. The sum of these two values is the location of the previous cyclic-shift. This function called the Last-to-first mapping (LF-mapping).

2.3.6 Dynamic suffix arrays

2.3.7 Incremental-parsing and error-recovery

2.4 Our contribution

This thesis will present and evaluate a tool which provides clone detection capabilities in a real-time IDE environment. The main goal will be to create an incremental tool which fits well into the development cycle and can efficiently update its results while writing code. Areas of focus will therefore be:

- Real-time / Incremental detection of code clones using dynamic suffix arrays
- IDE and language agnostic tooling such as LSP and Tree-sitter

2.4.1 Incremental clone detection using dynamic suffix arrays

The main area which we have explored is making the tool efficient in terms of incrementally updating whenever edits are performed in the editor. Most clone detection tools calculates clones from scratch and have no functionality to more efficiently calculate the clones after a small edit has been applied to the source code. Our tool utilizes dynamic suffix arrays to quickly update and find/remove clones, often faster than calculating the clones from scratch using a linear time suffix array construction algorithm.

We have also focused on how necessary information to calculate the clones are stored in memory in order to avoid too much memory usage, without loss in terms of accuracy of clones, or time spent calculating them.

2.4.2 LSP for IDE-based clone management

The tool gives programmers the ability to view clones in their IDE. We will utilize features of LSP such as diagnostics and code-actions, in order to provide clone management to any editor which implements the LSP protocol.

The following user-stories shows how interaction with the LSP server works.

- A programmer wants to see code clones for a file in their project, the programmer opens the file in their IDE and is displayed diagnostics in the code wherever there are detected clones. The matching code clones are not necessarily in the same file.
- A programmer wants to see all code clones for the current project. The programmer opens the IDEs diagnostic view and will see all code clones detected as diagnostics there. The diagnostic will contain information like where the clone exists, and where the matching clone(s) are.
- A programmer wants to jump to the corresponding match of a code clone in their editor. The programmer moves their cursor to the diagnostic and will see a list of the matching code clones. The programmer will select the wanted code clone which will move the cursor to the file and location of the selected code clone.
- A programmer wants to remove a set of clones by applying the "extract-method" refactoring. The programmer performs the necessary refactorings, saves the file and will get fast feedback whether the clones are now eliminated.

The interaction with the LSP server will depend on the client's implementation of LSP. If the LSP client is limited in its capabilities, meaning it does not implement the entire protocol, the tool will be limited in how the programmer can interact with it.

2.4.3 Evaluation

We will evaluate this tool based on different criteria, which combined will provide a basis for evaluating the tool as a whole.

Since the tool is focused on efficient detection and management of code clones, real-time performance of the tool will be a high priority in its evaluation. The tool will implement different techniques of detecting and merging clones. These will be empirically compared against each other. The tool will also be evaluated against existing tools empirically. We will utilize BigCloneBench [21] to evaluate detection techniques, by running our detection techniques in a standalone mode. We will distinguish between initial detection and incremental detection when evaluating.

The tool will also be evaluated in how well it fits into the software development cycle. Can we determine if this tool is an effective way to detect a clone early in its lifecycle so that they can be removed before it manifests in the source code? In relation to this, we will evaluate if LSP is a suitable tool for use in clone management and refactoring in general. Can LSP provide all the features one would want in a modern analysis tool? What is missing, and how could the LSP protocol be extended in order to facilitate this? We believe that if LSP is an appropriate tool to use for clone management, LSP will also be an appropriate tool for static analysis tools in general.

Chapter 3

Design

3.0.1 Architecture of tool

Figure 3.1 shows the architecture of the tool. The server communicates with the IDE and delegates the work of managing clones to the detection engine and the merge engine. The tool also stores an index of all source code files in the current project.

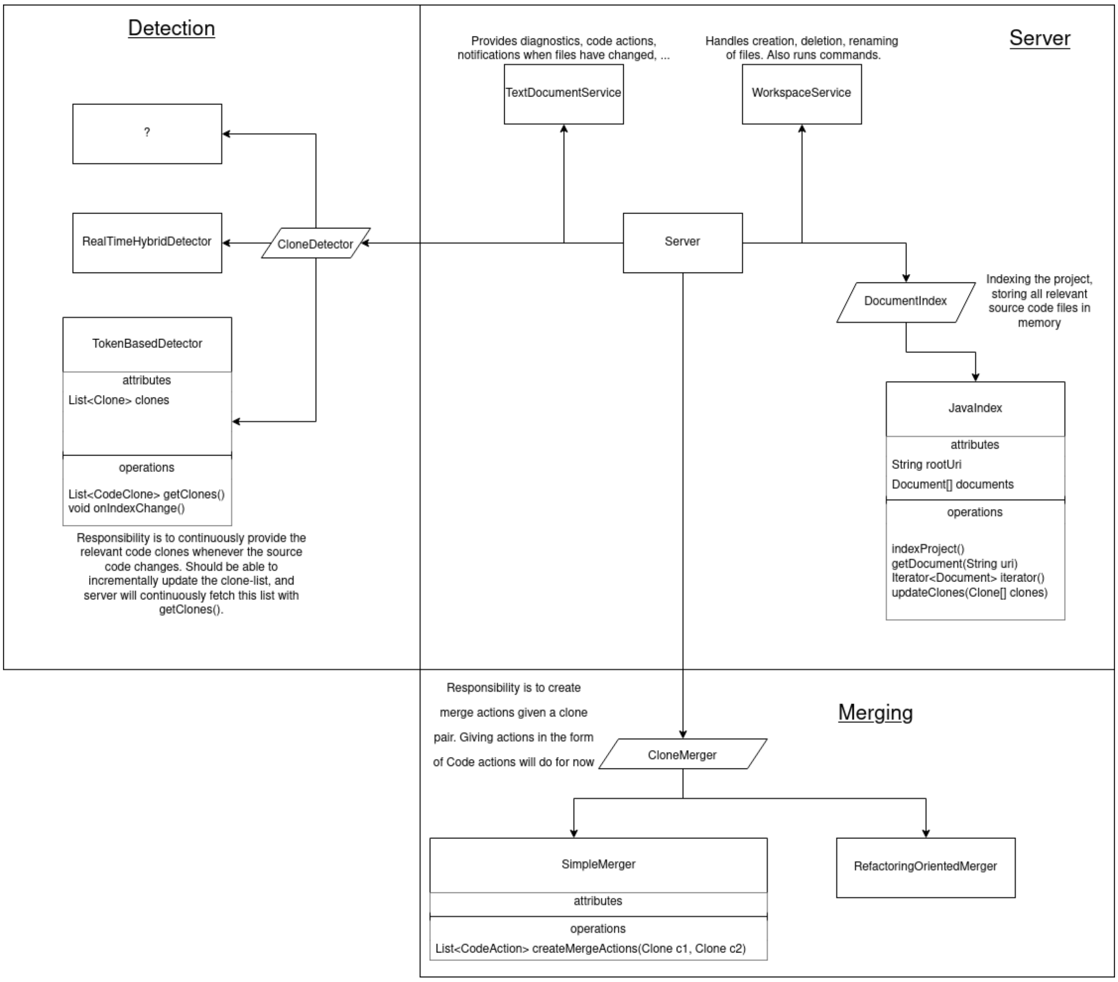

Figure 3.1: Tool architecture

Chapter 4

Evaluation

Look at how bad duplicate code is

Chapter 5

Conclusion

The end

Chapter 6

Future work

Ending duplicate code once and for all

Bibliography

[1] Raihan Al-Ekram et al. "Cloning by accident: an empirical study of source code cloning across software systems". In: *2005 International Symposium on Empirical Software Engineering (ISESE 2005), 17-18 November 2005, Noosa Heads, Australia.* IEEE Computer Society, 2005, pp. 376–385. DOI: 10.1109/ISESE.2005.1541846. URL: https://doi.org/10.1109/ISESE.2005.1541846.

[2] Paris Avgeriou et al. "Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162)". In: *Dagstuhl Reports* 6.4 (2016), pp. 110–138. DOI: 10.4230/DagRep.6.4.110. URL: https://doi.org/10.4230/DagRep.6.4.110.

[3] Brenda S. Baker and Raffaele Giancarlo. "Sparse Dynamic Programming for Longest Common Subsequence from Fragments". In: *Journal of Algorithms* 42.2 (2002), pp. 231–254. ISSN: 0196-6774. DOI: https://doi.org/10.1006/jagm.2002.1214. URL: https://www.sciencedirect.com/science/article/pii/S0196677402912149.

[4] Max Brunsfeld. *Tree-sitter.* https://tree-sitter.github.io/tree-sitter/. Accessed: 2022-04-16.

[5] Diego Cedrim et al. "Understanding the impact of refactoring on smells: a longitudinal study of 23 software projects". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017.* Ed. by Eric Bodden et al. ACM, 2017, pp. 465–475. DOI: 10.1145/3106237.3106259. URL: https://doi.org/10.1145/3106237.3106259.

[6] P.B. Crosby. *Quality is Free: The Art of Making Quality Certain.* New American Library, 1980. ISBN: 9780451624680. URL: https://books.google.no/books?id=3TMQt73LDooC.

[7] Martin Fowler. *Refactoring - Improving the Design of Existing Code.* Addison Wesley object technology series. Addison-Wesley, 1999. ISBN: 978-0-201-48567-7. URL: http://martinfowler.com/books/refactoring.html.

[8] Carlo Ghezzi and Dino Mandrioli. "Incremental Parsing". In: *ACM Trans. Program. Lang. Syst.* 1.1 (1979), pp. 58–70. DOI: 10.1145/357062.357066. URL: https://doi.org/10.1145/357062.357066.

[9] Nils Göde and Rainer Koschke. "Incremental Clone Detection". In: *2009 13th European Conference on Software Maintenance and Reengineering.* 2009, pp. 219–228. DOI: 10.1109/CSMR.2009.20.

[10] Benjamin Hummel et al. "Index-based code clone detection: incremental, distributed, scalable". In: *2010 IEEE International Conference on Software Maintenance.* 2010, pp. 1–9. DOI: 10.1109/ICSM.2010.5609665.

[11] Katsuro Inoue. “Introduction to Code Clone Analysis”. In: *Code Clone Analysis: Research, Tools, and Practices*. Ed. by Katsuro Inoue and Chanchal K. Roy. Singapore: Springer Singapore, 2021, pp. 3–27. ISBN: 978-981-16-1927-4. DOI: 10.1007/978-981-16-1927-4_1. URL: https://doi.org/10.1007/978-981-16-1927-4_1.

[12] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. 2nd. USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0201729156.

[13] Juha Kärkkäinen. “Suffix Array Construction”. In: *Encyclopedia of Algorithms*. Ed. by Ming-Yang Kao. New York, NY: Springer New York, 2016, pp. 2141–2144. ISBN: 978-1-4939-2864-4. DOI: 10.1007/978-1-4939-2864-4_412. URL: https://doi.org/10.1007/978-1-4939-2864-4_412.

[14] Shinji Kawaguchi et al. “SHINOBI: A Tool for Automatic Code Clone Detection in the IDE”. In: *16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France*. Ed. by Andy Zaidman, Giuliano Antoniol, and Stéphane Ducasse. IEEE Computer Society, 2009, pp. 313–314. DOI: 10.1109/WCRE.2009.36. URL: https://doi.org/10.1109/WCRE.2009.36.

[15] Liuqing Li et al. “CCLearner: A Deep Learning-Based Clone Detection Approach”. In: *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 2017, pp. 249–260. DOI: 10.1109/ICSME.2017.46. URL: https://doi.org/10.1109/ICSME.2017.46.

[16] Tung Thanh Nguyen et al. “Scalable and incremental clone detection for evolving software”. In: *25th IEEE International Conference on Software Maintenance (ICSM 2009), September 20-26, 2009, Edmonton, Alberta, Canada*. IEEE Computer Society, 2009, pp. 491–494. DOI: 10.1109/ICSM.2009.5306283. URL: https://doi.org/10.1109/ICSM.2009.5306283.

[17] Ge Nong, Sen Zhang, and Wai Hong Chan. “Two Efficient Algorithms for Linear Time Suffix Array Construction”. In: *IEEE Trans. Computers* 60.10 (2011), pp. 1471–1484. DOI: 10.1109/TC.2010.188. URL: https://doi.org/10.1109/TC.2010.188.

[18] Chaiyong Ragkhitwetsagul and Jens Krinke. “Siamese: scalable and incremental code clone search via multiple code representations”. In: *Empir. Softw. Eng.* 24.4 (2019), pp. 2236–2284. DOI: 10.1007/s10664-019-09697-7. URL: https://doi.org/10.1007/s10664-019-09697-7.

[19] M. Rieger, S. Ducasse, and M. Lanza. “Insights into system-wide code duplication”. In: *11th Working Conference on Reverse Engineering*. 2004, pp. 100–109. DOI: 10.1109/WCRE.2004.25.

[20] Chanchal Kumar Roy, James R. Cordy, and Rainer Koschke. “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach”. In: *Sci. Comput. Program.* 74.7 (2009), pp. 470–495. DOI: 10.1016/j.scico.2009.02.007. URL: https://doi.org/10.1016/j.scico.2009.02.007.

[21] Jeffrey Svajlenko and Chanchal K. Roy. “BigCloneBench”. In: *Code Clone Analysis*. Ed. by Katsuro Inoue and Chanchal K. Roy. Springer Singapore, 2021, pp. 93–105. DOI: 10.1007/978-981-16-1927-4_7. URL: https://doi.org/10.1007/978-981-16-1927-4%5C_7.

[22] Md Sharif Uddin, Chanchal K. Roy, and Kevin A. Schneider. “Towards Convenient Management of Software Clone Codes in Practice: An Integrated Approach”. In: *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*. CASCON ’15. Markham, Canada: IBM Corp., 2015, pp. 211–220.

[23] Esko Ukkonen. “On-line construction of suffix trees”. In: *Algorithmica* 14.3 (1995), pp. 249–260.

[24] Tim A. Wagner. “Practical Algorithms for Incremental Software Development Environments”. PhD thesis. EECS Department, University of California, Berkeley, Mar. 1998. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/1998/5885.html`.

[25] Zhipeng Xue et al. “SEED: Semantic Graph based Deep detection for type-4 clone”. In: *CoRR* abs/2109.12079 (2021). arXiv: `2109.12079`. URL: `https://arxiv.org/abs/2109.12079`.

[26] Minhaz F. Zibran and Chanchal K. Roy. “IDE-Based Real-Time Focused Search for near-Miss Clones”. In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. SAC ’12. Trento, Italy: Association for Computing Machinery, 2012, pp. 1235–1242. ISBN: 9781450308571. DOI: `10.1145/2245276.2231970`. URL: `https://doi.org/10.1145/2245276.2231970`.