UNIVERSITY
OF OSLO

Master's thesis

CCDetect-LSP: Incremental code clone detection for IDEs

Language- and IDE agnostic incremental code clone detection for IDEs

Jakob Konrad Hansen

60 ECTS study points

Department of Informatics
Faculty of Mathematics and Natural Sciences

Spring 2023

Jakob Konrad Hansen

CCDetect-LSP: Incremental code clone detection for IDEs

Language- and IDE agnostic incremental code clone detection for IDEs

Abstract

Duplicated code is bad

Contents

Chapter 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 [10, p. 9]. Refactoring 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 programmers tend to refactor smelly code, they are rarely successful at eliminating the smells they are targeting [8]. Most refactorings performed were either "smell-neutral", meaning that the code smell is not eliminated, or "stinky", meaning that they introduced more code smells than they eliminated. Automated tools that help programmers 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 (duplicate code analysis) has in the last decade become a highly active field of research and many tools have been developed to detect duplicated code [14, p. 6].

1.1 Motivation and problem statement

Many tools and algorithms exist for duplicate code detection. However, few of these have the capability of efficiently detecting duplicated code in a real-time IDE environment. Incremental algorithms that do not recompute all clones from scratch are interesting for use-cases such as IDEs and different Git revisions of the same source code, but this type of algorithm has not been thoroughly explored for code clone analysis.

The current landscape of clone detection algorithms and tools is therefore lacking in terms of integration and support for fast incremental updates within an IDE. This limits the ability for programmers to easily detect duplicated code as they work, as these tools are also often limited to specific programming languages or IDEs.

Our proposed solution addresses this issue by introducing a new algorithm that is capable of detecting and updating all existing code clones in real-time as source code changes, faster than redoing the analysis from scratch. The tool which implements this algorithm is also programming

language and IDE agnostic, which allows programmers to seamlessly incorporate the detection of duplicated code into their existing development environment.

1.2 Structure

The current chapter describes the motivation and contribution of the thesis. Chapter 2 continues by giving some background on code clone analysis, some existing tools and some preliminary algorithms and data structures used in the implementation. Chapter 3-5 describes the implementation of the tool and the algorithms used for initial and incremental clone detection. In chapter 6, the tool is evaluated based on different criteria, and compared to other existing solutions. Chapter 7 discusses the results of the evaluation and discusses some choices made in the implementation. Chapter 8 concludes the thesis and lists future work.

1.3 Our contribution

This thesis will present and evaluate a tool which provides clone detection capabilities in a real-time IDE environment. The main goal is to create a tool which fits well into the development cycle and can efficiently update its analysis while writing code.

The tool will allow the user to list and interact with all the code clones in the codebase, jump between matching clones, and get quick feedback while editing code in order to determine which clones are introduced/eliminated.

Existing incremental clone detection tools either do not fit into an IDE scenario (techniques based on distributed computing), are limited in terms of what clones they display, or have not been shown to scale well in terms of processing time or memory usage for larger codebases. Therefore, this thesis will focus on the following areas.

Incremental clone detection: The main focus of this thesis is making the tool efficient in terms of incrementally updating whenever edits are performed in the IDE. 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 implements a novel application of dynamic suffix arrays to quickly update and add/remove clones, often faster than calculating the clones from scratch using a linear time suffix array construction algorithm.

IDE and language agnostic clone detection: The tool gives programmers the ability to view clones in their IDE. Utilizing features of the language server protocol (LSP) such as diagnostics and code-actions, the tool provides clone analysis to any editor which implements the LSP protocol. As far as we are aware there are no other clone detection tools which utilizes the LSP protocol to provide clone analysis. The tool is also language agnostic in the sense that it only needs a grammar for a language to analyze it.

Chapter 2

Background

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" [9, p. 8]. In software, a simple measure of "conformance to requirements" is correctness, and a lack of bugs. However, software quality is often measured in other metrics, including metrics which are not directly related to functionality [17, 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 [2]. Therefore, research into tools and techniques which can assist in reducing duplicated code will be of benefit to almost all software.

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 [3, p. 111]. An example of this, in the context of duplicated code, is the "Shotgun-Surgery" [10, 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 Code clones

Duplicated code is often described as "code clones", as a pair of code snippets which are duplicated are considered clones of each other.

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.*

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

$$a \overset{clone}{\longleftrightarrow} b \overset{clone}{\longleftrightarrow} c$$

where a, b, c are code snippets and $\overset{clone}{\longleftrightarrow}$ 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 clones are generally classified into four types [14]. 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 syntactically 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:

TODO: Redo these

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

```
for (int i = 0; i < 10; i++) {
    // A comment

    print(i);
}
```

Type-2 clones are structurally identical. Possible differences include identifiers, literals and types. Type-2 clones are not much harder to detect than type-1 clones, since consistently renaming identifiers, literals and types allow a type-1 detection algorithm to find type-2 clones [33]. This type of clone is relevant to consider in refactoring scenarios when merging code clones. This is because type-2 clones can be relatively simple to parameterize in order to hide the differences in the merged code, and allows the original locations of clones to be replaced with a call to the merged code, with different parameters. Example:

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

```
for (int (*\textbf j*) = (*\textbf 1*); (*\textbf j*) < 10; (*\textbf j
    print((*\textbf j*));
}
```

Type-3 clones are required to be structurally similar, but not equal. Differences include statements that are added, removed or modified. This clone type relies on a threshold θ which determines how structurally different snippets can be in order to be considered Type-3 clones [14]. 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, but are generally only relevant to detect when they have similar functionality. Detecting this type of clone is very challenging, but attempts have been made using program dependency graphs [32]. The following example shows two code snippets which have no clear syntactic or structural similarity, but is functionally equal:

```
print((n*(n-1))/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 [33, p. 1].

2.3 Code clone detection process and techniques

The Code clone detection process is generally split into (but is not limited to) a sequence of steps to identify clones [14]. 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. An algorithm could potentially do multiple transformation before arriving at the wanted representation

   (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. **Source-mapping / 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

Matching techniques are techniques which can be applied to source-code to detect clone-pairs. Most matching technique will also require specific pre-processing to be done in the earlier steps, for example building an AST. Some of the most explored techniques are as follows [25]:

Text-based approaches do little processing of 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 [4]. A common approach to detect clones using token-streams is with a suffix-tree[33]. A suffix-tree can solve the *Find all maximal repeats* problem efficiently, which in essence is the same problem as finding clone pairs. A similar algorithm can also be performed using a suffix-array instead, which requires less memory. This type of code clone detection is very fast compared to more intricate types of matching techniques.

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 compare subtrees with a tolerant tree matching algorithm for detecting type-3 clones [5]. Variable names, literal values and types 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 [16].

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 [33].

2.4 Incremental clone detection

Incremental clone detection involves avoiding recalculation of already calculated results when performing code clone detection. Since most code of a codebase will not change between revisions, a lot of processing can be avoided. However, this is not a simple problem, since changes in a single location can affect clone detection results across the entire codebase.

In order to incrementally detect code clones, an algorithm is run which calculates the initial clones, and for successive revisions of the source code, this list is incrementally updated, more efficiently than the initial run. Different algorithms will also maintain different data structures to support detecting new clones faster in successive revisions.

Göde and Koschke proposed the first incremental clone detection algorithm [12]. 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 requires a substantial amount of memory, and is therefore limited in scalability.

Nguyen et al. [21] 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. [13] 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. This technique uses distributed computing to speed up its detection process.

More recently, Ragkhitwetsagul and Krinke [23] 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 return 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.

2.5 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 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 maintain it.

Ghezzi and Mandrioli introduced the notion of incremental parsing, with 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 [11].

Tim A. Wagner et al. [31] 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 [6]. These features combined allow Tree-sitter to become a powerful tool for editing and has been used for IDE features such as syntax-highlighting, refactoring and code navigation.

2.6 Code clone IDE tooling

Developers are not always aware of the creation of clones in their code. *Clone aware development* means including clone management as a part of the software development process. Clone aware development therefore requires programmers to be aware of and be able to identify code clones while programming. Since large software projects can contain a lot of duplication, it can be hard to keep track of and manage clones. Tools which help developers locate and deal with clones can be a solution to this. 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" [24, 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 multiple existing clone detection tools, and the following section will go over tools that are integrated into an IDE and offer services to the programmer while developing. Some of these tools will

The IDE-based tools which exist can be categorized as follows [28, 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.

The following tools have been developed as IDE tools to allow for clone-aware development:

- 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 [33]. It was later used in the tool *SimEclipse* [28] 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, but the clones being displayed seem to only be clones of the source-code which is currently being edited. It can detect Type-1 and Type-2 code clones and uses a token-based suffix array approach to detect clones. The paper does not describe how the suffix array is updated, but as the paper was released before the first paper describing dynamic suffix arrays[26], one can assume that the suffix array is not dynamically updated. [19].

- 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.

2.7 The Language Server Protocol

Static analysis tools that integrate with IDEs are usually tightly coupled to a specific IDE and its APIs, like parsing and refactoring support. This makes it difficult 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 is useful if such tools could be made IDE agnostic. The Language Server Protocol (LSP) is a protocol which attempts to solve this problem [20].

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.

2.8 Preliminary algorithms and data structures

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

Incremental-parsing

In our detection algorithm we will need to be able to parse our source code in order to select specific syntactic regions and extract the tokens.

...

Suffix trees

A classic algorithm [33, 12] for code clone detection traverses a suffix-tree in order to find maximal repeats in all suffixes of the input string T.

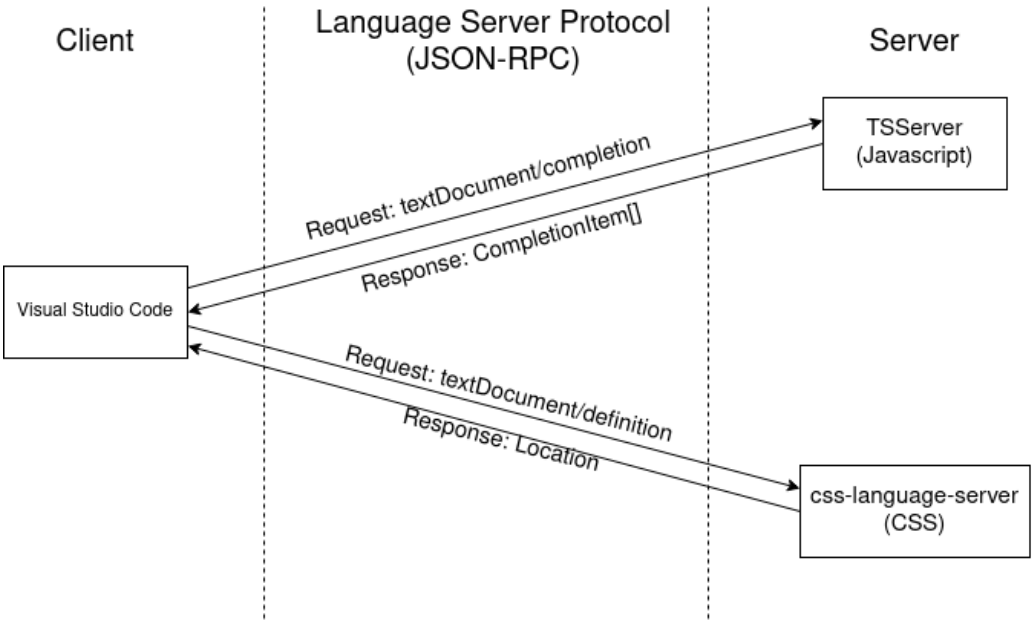

Figure 2.1: Example client-server interaction using LSP

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. A common usage of a suffix tree is to determine whether or not a suffix exists in T, and where in T the suffix starts.

Figure 2.2 shows the suffix-tree for T = BANANA\$. In order to determine where the suffix ANANA\$ exists in T, one can start from the root, and traverse the tree, choosing the child node which correspond to the next character of the suffix which has not been "matched" yet.

$$\text{root} \xrightarrow{A} \text{node} \xrightarrow{\text{NA}} \textit{node} \xrightarrow{\text{NA\$}} 1$$

Following this path, we see that the suffix ANANA\$ exists in T at index 1.

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, lowering the complexity [29].

This data structure also 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. The problem of finding all maximal repeats can be solved with a suffix tree using the following theorem:

Theorem 1 (Repeats in suffix tree)**.** *Every internal node (except for the root) 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.*

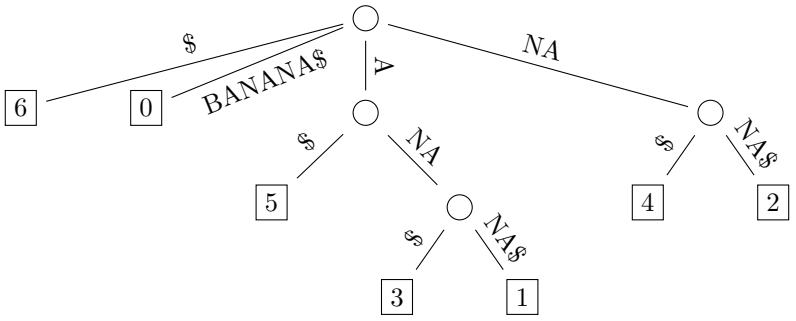

Figure 2.2: Suffix tree for T = BANANA$

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 maximal repeats would find the internal nodes which represents the longest strings.

The classic algorithm [33, 12] 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 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 subtree rooted in the internal node, which each hold 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 contain or overlap each other.

In figure 2.2, there are three repeats, ANA, A and NA. The only maximal repeat would be ANA, since A and NA are not maximal.

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 the suffix at a given position has. The rank of a suffix is its lexicographical ordering in T. 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 lexicographically smallest suffix in T.*

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 is an array which describes how many common char-

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	0
1	5	3	0
2	3	6	1
3	1	2	3
4	0	5	0
5	4	1	0
6	2	0	2

(c) SA, ISA and LCP

Table 2.1: T = BANANA$

acters there are in a prefix between two suffixes which are next to each other in 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 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 maximal common prefix of length n. LCP[0] is undefined or 0.*

For example, in table 2.1, the LCP value at position 3 contains the number of characters in the longest-common prefix of suffixes starting at position 1 (ANANA$) and position 3 (ANA$). These suffixes have 3 common characters in their prefix, therefore the LCP value at position 3 is 3.

Suffix arrays can be constructed in linear time in terms of the length of T. Many suffix array construction algorithms (SACA) have been discovered in the last decade[18], many of which run in linear-time. An algorithm which has been shown to be very efficient in practice is Nong and Chan's algorithm based on recursive suffix sorting of smaller strings [22].

The enhanced (extended) suffix array is the suffix array and the additional LCP array, which has been shown to be as powerful as a suffix tree, in terms of what can be computed with it with the same time complexity, but uses a smaller amount of memory [1].

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 [15].

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

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

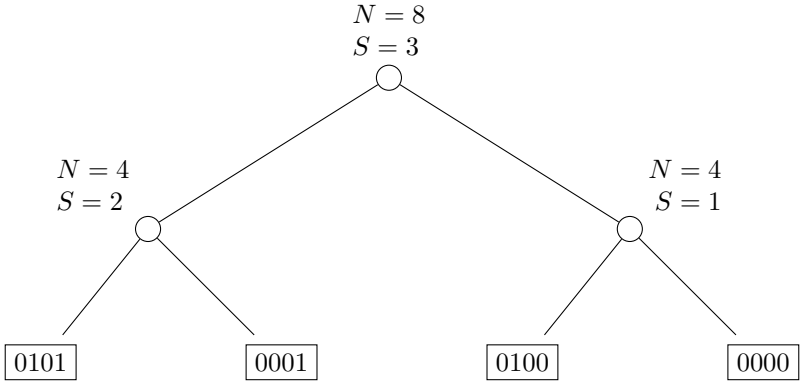

Figure 2.3: Dynamic bitset

Wavelet trees and wavelet matrices

Burrows-Wheeler transform

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

L	F
0	$Rank_A(0) + C[A] = 0 + 1 = 1$
1	$Rank_N(1) + C[N] = 0 + 5 = 5$
2	$Rank_N(2) + C[N] = 1 + 5 = 6$
3	$Rank_B(3) + C[B] = 0 + 4 = 4$
4	$Rank_\$(4) + C[\$] = 0 + 0 = 0$
5	$Rank_A(5) + C[A] = 1 + 1 = 2$
6	$Rank_A(6) + C[A] = 2 + 1 = 3$

(c) LF-mapping

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

The Burrows-Wheeler transform (BWT) is a transform on strings, often performed on strings to improve compression [7]. 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. $ is added to the string as a unique terminating character, this makes some of the algorithms on the BWT simpler. $ is always the smallest character lexicographically. Table 2.2 shows the BWT for the string S = BANANA$.

There is a direct correlation between the suffix array of S and the BWT of S. Table 2.2 also shows that the indices of the sorted cyclic-shifts correspond to exactly the SA of S. 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 S. Algorithm 2 shows how the BWT of S can be calculated directly from the SA of S in linear time. Since this is a one to one 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 implementation chapter for the dynamic code clone detection algorithm.

Algorithm `ComputeBWT(`*S, SA*`)`

```
n ← S_len
BWT ← string of length n
for i from 0 to n do
    pos ← (SA[i] − 1) % n
    BWT[i] = §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).

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 previous cyclic-shift). The character in the last column of that cyclic-shift is $T[n-2]$. If there are multiple of the same character, the nth occurrence of a certain symbol in the last column will map to the nth occurrence of the same symbol 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.

We can use the fact that the first column consists of the letters of T in sorted order to determine the previous cyclic-shift with only the last column. We can calculate the previous 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 is called the Last-to-first mapping (LF-mapping). Calculating the LF-mapping can be done efficiently by using a rank/select data structure to calculate the rank of all the characters in the BWT, and an array C which contains the number of occurrences of each letter in the BWT.

The LF-mapping will also be useful in the detection algorithm when dynamically updating the suffix array.

Dynamic suffix arrays

Chapter 3

Implementation: LSP server architecture

The tool is integrated into IDEs via the Language server protocol. The goal is to give users of the tool an overview of all clones as they appear in source-code.

The tool shows error-messages (diagnostics in LSP terms) which indicate which section of code a clone covers, they also provide information about the matching clone and a code-action to navigate to it.

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. Alternatively, a code-action can be invoked to navigate, if the client does not implement the `DiagnosticRelatedInformation` interface.
- 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 quick feedback whether the clones are now gone.

TODO: Redo this figure

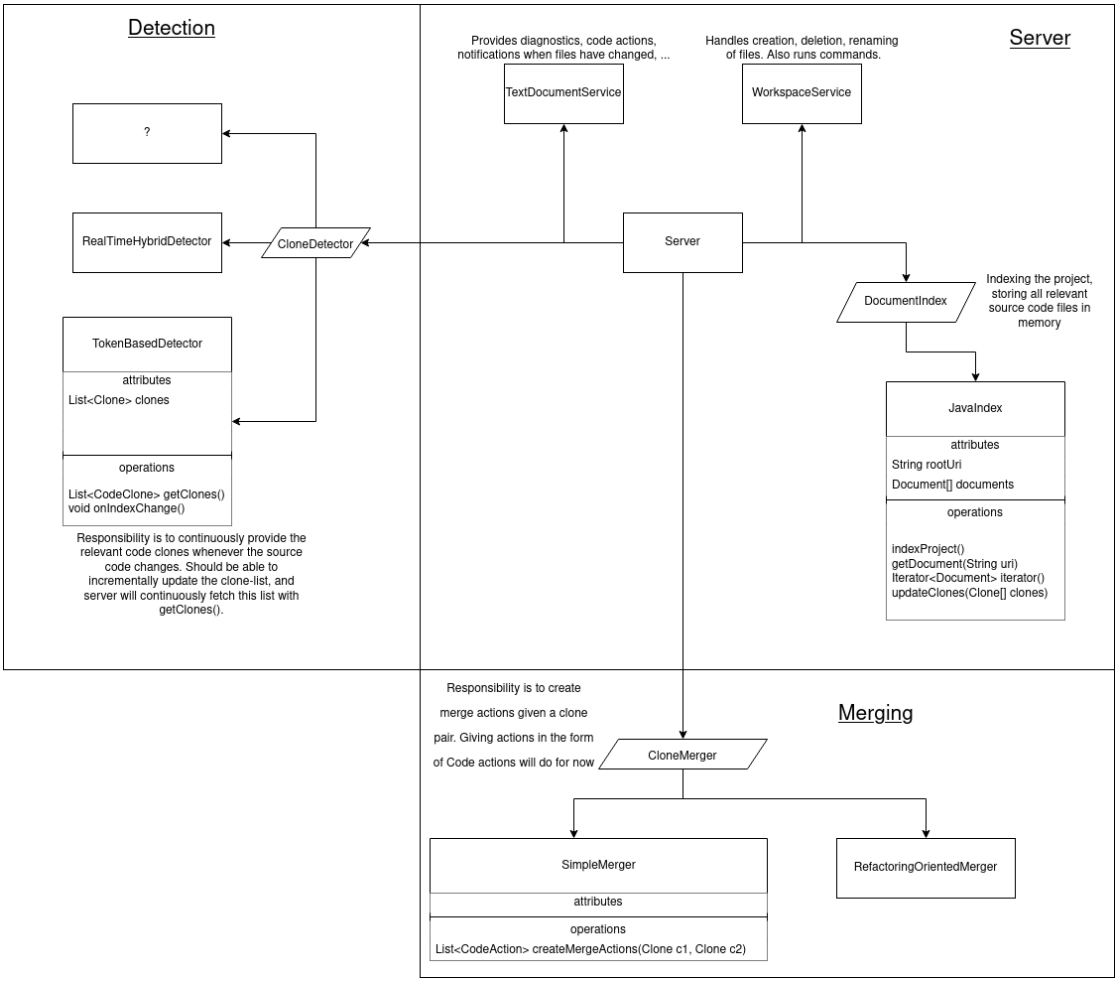

Figure 3.1: Tool architecture

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.

3.1 Document index

Upon starting, the LSP server requires indexing of the project for conducting analysis. This involves creating an index and inserting the relevant documents. A document contains the content of a file along with extra information such as the file's URI and some information which is useful for the later clone detection. We define the following interface for our documents:

```
interface Document {
    String uri,
    String content,
    AST ast,

    // Location in fingerprint
    int start,
    int end

    // Used for incremental updates
    int[] fingerprint,
    boolean open,
    boolean changed,
}
```

Each document in the index primarily consists of the contents, the URI and the AST of the document in its current state. Storing the AST will be useful for the incremental detection algorithm.

There are two things to consider when determining which files should be inserted into the index. First, we are considering only files of a specific file type, since the tool does not allow analysis of multiple programming languages at the same time. Therefore, the index should contain for example only `*.java` files if Java is the language to analyze. Secondly, all files of that file type might not be relevant to consider in the analysis. This could for example be generated code, which likely contains a lot of duplication, but is not practical or necessary to consider as duplicate code, since this is not code which the programmer interact with directly. Therefore, the default behavior is to consider only files of the correct file type, which are checked into Git. The tool supports adding all files in a folder, or all files checked into Git.

When a document is first indexed by the server, the file contents is read from the disk. However, as soon as the programmer opens this file in their IDE, the source of truth for the files content is no longer on the disk, as the programmer is changing the file continuously before writing to the disk. The LSP protocol defines multiple RPC messages which the client sends to the server in order for the server to keep track of which files are opened, and the state of the content of opened files.

Upon opening a file, the client will send a `textDocument/didOpen` message to the server, which contains the URI for the opened file. The index will at this point set the flag `isOpened` for the relevant document and stops reading its contents from disk. Instead, updates to the file are obtained via the `textDocument/didChange` message. This message can provide either the entire

content of the file each time the file changes, or it can provide only the changes and the location of the change. Receiving only the changes will be useful for this algorithm when the analysis incrementally reparses the document.

3.2 Displaying and interacting with clones

Chapter 4

Implementation: Initial detection

This chapter discusses the detection module of the tool. It consists of the detection algorithm which takes the document index as input, and outputs a list of code clones. The initial input to the algorithm will be the raw source code of each document in the index, in text format.

The algorithm detects syntactical type-1 code clones, based on a token-threshold. Clones detected are therefore snippets of source-code which has at least N equal tokens, where N is a configurable parameter. There are also two types of clones which are filtered:

- Clones which are completely contained within another larger clone.
- Clones which start in one fragment and ends in the next are cut off at the end of the first fragment.

In broad strokes, the algorithm first selects the relevant parts of source code to detect code clones in (fragment selection), then transforms the selected fragments into a smaller representation (fingerprinting). For the matching, an extended suffix array for the fingerprint is constructed, where the LCP array is used to find matching instances of source-code. Finally, clones are filtered and aggregated into clone classes before they are source-mapped back to the original source-code locations, which the LSP server can display as diagnostics.

4.1 Fragment selection

The first phase of the algorithm considers how to extract the relevant fragments of source code which should be considered for detection. A fragment in this case is considered as a section of abstract syntax, meaning that a particular type of node in the AST and the tokens it encompasses should be extracted. Since the algorithm is language agnostic, it is not feasible to have a single algorithm for fragment extraction or to define a separate fragment extraction algorithm for every possible language. Therefore, the tool allows the user to define a fragment query via tree-sitter queries[6]. Tree-sitter queries are flexible queries to extract specific nodes or subtrees in an AST. For example, in Java it would be natural to consider only methods. The tree-sitter query for selecting only the method nodes in a Java programs AST would be:

(method_declaration @method) (4.1)

This tree-sitter query selects the node with type `method_declaration` and "captures" it with the `@method` name, for further processing by the program. Readers interested in the details of the query system are referred to the tree-sitter documentation[6].

Implementing something similar for another parser/AST could be as simple as traversing the tree until a node of a specific type is found, using the visitor pattern.

Using a tree-sitter query, the algorithm parses the program into an AST, queries the tree for all the nodes which matches the query (taking care not to capture nodes which are children of already captured nodes) and extracts all the tokens which the node covers.

4.2 Fingerprinting

```
public class Math() {
    public int multiplyByTwo(int param) {
        return param * 2;
    }

    public int addTwo(int param) {
        return param + 2;
    }
}
```

(a) Source-code

Token	Fingerprint
public	2
int	3
multiplyByTwo	4
(	5
param	6
)	7
{	8
return	9
*	10
2	11
;	12
}	13
addTwo	14
+	15

(b) Fingerprint mapping

[2, 3, 4, 5, 3, 6, 7, 8, 9, 6, 10, 11, **1**, 2, 3, 14, 5, 3, 6, 7, 8, 9, 6, 15, 11, **1**, **0**]

(c) Fingerprint, terminals highlighted

Figure 4.1: Example fingerprint of Java source-code

The next phase of the algorithm is to transform the extracted source code into a representation which is less computationally heavy for the matching algorithm. The goal is to reduce the total size of the input which needs to be processed by the matching algorithm.

Since the algorithm that is used for matching is based on a suffix array, the representation should be in a format similar to a string, which is the standard input to a suffix array construction algorithm. However, it's not strictly necessary to use strings. The essential property of the input array is that each element is comparable. Therefore, we can use an array of integers instead, which is preferable, since there are often a lot more unique integers available than unique characters

in a programming language. The algorithm will need to have a lot of unique elements in the representation because each unique token value should be represented by one unique element, therefore integers are a good choice.

The algorithm utilizes fingerprinting in order to reduce the size of the representation. Fingerprinting is a technique which involves taking some part of the input and mapping it to a smaller bit string, which uniquely identifiers that part. In this case, the algorithm takes each token of the source-code, and maps it to an integers bit string. Each unique token value (tokenized by tree-sitter) is mapped to unique integer values. Note that the algorithm maps token values, not types. This ensures that tokens of different variable-names and literals are not seen as equal. Figure 4.1 shows how a sample Java program could theoretically be fingerprinted. The fingerprint mapping starts its "count" at 2 to make space for some terminating characters. Each fragment is terminated by a 1 and the fingerprint is ultimately terminated by a 0. Having these values in the fingerprint will be useful for the matching algorithm where the suffix array is constructed and utilized for detection.

The fingerprint of each fragment is stored in a list which the relevant document object contains.

4.3 Suffix array construction

The next step is to input the fingerprint into a suffix array construction algorithm (SACA), so that the suffix array can be used to find maximal repeats. All the fingerprints which are stored in each document object are concatenated to be stored in a single integer array (with terminators after each fragment), which the suffix array (SA), inverse suffix array (ISA) and longest-common prefix array (LCP) is computed from.

We have utilized a straight-forward implementation of the "Induced sorting variable-length LMS-substrings" algorithm [22] which computes a suffix array in linear time. The following section will give high-level overview of how the algorithm works.

The algorithm will be explained with a string input, as this is most common for suffix arrays, and working with strings can be more clear to the reader when looking at suffixes. The algorithm is still applicable to our fingerprint, since an array of integers will work similarly to a string when given as input.

The "Induced sorting variable-length LMS-substrings" algorithm (often abbreviated SA-IS) is an algorithm that works by divide and conquering the array and inducing how to sort the bigger string from the smaller string, where the smaller string consists of the "building blocks" of the bigger string. First, we will introduce some definitions and theorems used for the algorithm. Input to the algorithm is a string S with length n. Let suffix(S, i) be the suffix in S starting at position i.

Definition 10 (L-type and S-type suffixes)**.** *A suffix starting at position i in a string S is considered to be L-type if it is lexicographically larger than the next suffix at position $i + 1$. Meaning that* suffix$(S, i) >$ suffix$(S, i + 1)$*. Conversely, a suffix is considered to be S-type if* suffix$(S, i) <$ suffix$(S, i + 1)$*. The sentinel suffix (\$) of S is always S-type.*

Note that two suffixes cannot be lexicographically equal, therefore all cases are handled by this definition. Determining the type of each suffix can be done in $O(n)$ by scanning S from right-to-left and observing the following properties: suffix(S, i) is L-type if $S_i > S_{i+1}$. Similarly,

suffix(S, i) is S-type if $S_i < S_{i+1}$. If $S_i = S_{i+1}$, then suffix(S, i) is the same value as suffix$(S, i+1)$. This is explained by the fact that if the first character of the current suffix is not equal to first character of the next suffix, we already know the type based on the first character. If the first character is equal however, we have effectively reduced the case to the next suffix, since we are now comparing the second character of the current suffix, with the next character of the second suffix. Since we have already computed the type of that suffix, we can reuse the value. Figure 3 shows an algorithm which determines the type of each suffix in a string in $O(n)$ time.

Algorithm `SuffixTypes(`S`)`

```
n ← S_len
Stype ← True
Ltype ← False

types ← Bitset of size n

Set(types, n, Stype)
Set(types, n − 1, Ltype)

for i from n − 2 to 0 do
    if S_i < S_{i+1} then
        Set(types, i, Stype)
    else if S_i > S_{i+1} then
        Set(types, i, Ltype)
    else
        Set(types, i, Get(types, i + 1))
    end
end
return types
```

Algorithm 3: Compute suffix types of a string

Definition 11 (LMS character)**.** *An LMS (Left-most S-type) character in a string S is a position i in S such that $S[i]$ is S-type and $S[i-1]$ is L-type. An LMS-suffix is a suffix in S which begins with an LMS character. The final character of S (the sentinel) is always S-type and the first character is always L type*

Definition 12 (LMS-substring)**.** *An LMS-substring in a string S is a substring $S[i..j]$ in S such that $i \neq j$, $S[i]$ and $[j]$ are LMS-characters and there are no other LMS-character between. The sentinel character is also an LMS-substring and is the only LMS-substring of length ≤ 3*

LMS-substrings form "basic-blocks" in the string S. Each LMS-substring overlap on two characters, and each substring should be monotonically decreasing, because of the type. Using this notion, we can sort all LMS suffixes recursively with the following theorems:

Theorem 2. *Given sorted LMS-suffixes of S, the rest of the suffix array can be induced in linear time.*

Theorem 3. *There are at most $n/2$ LMS-substrings in a string S of length n.*

The details of the theorems are left out, but the intuition of the second theorem will be explained

as it is the essential part of how the recursion allows us to build the suffix array in linear time.

We can construct a smaller string S_1 by first sorting the LMS-substrings. Sorting LMS-substrings can be done in linear time with a three-pass operation (details left out). After sorting, each equal LMS-substring is given a unique increasing integer value. Two LMS-substrings are considered equal if they are equal in terms of length, characters and types. The smaller string is now built by mapping each LMS-substring in the original ordering to its unique value, and forming a smaller string of it. of S This smaller string will be the reduced case which is used in the recursion. The string is at most $n/2$ in size, meaning there will be at most $\log_2(n)$ recursive calls. The recursive call will return the suffix array of the S_1. [18] proves a theorem which shows that sorting S_1 is equivalent to sorting the LMS suffixes of S. Therefore, the suffix array of S_1 can be mapped to the LMS suffixes of S, which is a subset of the SA of S. Now that we have

The base-case of the recursion is when there is one LMS-substring, the sentinel character. Intuitively, it is trivial to compute the suffix array of S when there is only one LMS-substring, since all suffixes except the sentinel are either in lexicographically increasing or decreasing order. When the recursive call returns, the SA of S_1 is used in another three-pass operation where the LMS suffixes guide the rest of the suffixes to their correct location in SA.

There will be $O(\log_2(n))$ recursive call, where each recursive call takes $O(n)$ time, with n halving in each call. Therefore, the recurrence will have the complexity of:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1. \\ T(n/2) + O(n) & \text{if } n > 1. \end{cases} = O(n)$$

Building ISA and LCP arrays

Computing the ISA after constructing the SA is trivial. Since the ISA is simply the inverse of SA, it can be constructed in linear time with a single loop as seen in Algorithm 4

Algorithm `ComputeISA(`SA`)`
```
n ← SA_len
ISA ← array of size n
for i from 0 to n do
    ISA[SA[i]] ← i
end
return ISA
```

Algorithm 4: Compute ISA from SA

Computing the LCP in linear time is more complicated and requires some thought about which order to insert LCP values. We will also add one extra restriction to the LCP values, being that the LCP values cannot match past a 1, which were the terminal value between fragments. This restriction will be useful when we want to extract clones using the LCP. The algorithm to compute the LCP is shown in Algorithm 5. The intuition for this algorithm is that if a suffix at position i which has an LCP value l describing the common-prefix between it and some other suffix at position j, then the LCP value of the suffix at position $i + 1$ is at least $l - 1$, since the suffix at $i + 1$ and $j + 1$ is the same suffix as the suffixes at position i and j, with the first

character cut off. Therefore, they share at least $l - 1$ characters, and the algorithm can start comparing the characters at that offset.

Algorithm `ComputeLCP(`*S, SA, ISA*`)`

```
n ← SA_len
LCP ← array of size n
lcpLen ← 0
for i from 0 to n − 1 do
    r ← ISA[i]
    prevSuffix ← SA[r − 1]
    while S[i + lcpLen] = S[prevSuffix + lcpLen] and S[i + lcpLen] ≠ 1 do
        lcpLen ← lcpLen + 1
    end
    LCP[r] ← lcpLen
    lcpLen ← Max(0, lcpLen − 1)
end
return ISA
```

Algorithm 5: Compute LCP from input string S, SA, and ISA

4.4 Clone extraction

With the extended suffix array structure, we can now consider which substrings (prefixes of suffixes) we want to extract as potential code clones. In this phase the indices of the fingerprint which we consider to be code clones are extracted, which will in the next phase be used to map back to the original source code.

A solution is to extract every suffix which has an LCP value which is greater than the token threshold (except for the suffix at index 0 in SA). The algorithm would be a single loop over S, using ISA to find the corresponding LCP value. This finds the clone indices, as shown in Algorithm 6.

Algorithm `SimpleCloneExtraction(`*S, ISA, LCP*`)`

```
n ← S_len
clones ← list
for i from 0 to n − 1 do
    if ISA[i] = 0 then
        continue
    end

    if LCP[ISA[i]] ≥ THRESHOLD then
        Insert(clones, i) // Adds i to the clone-list
    end
end
return clones
```

Algorithm 6: Extract clones indices in a string S

However, this algorithm will return a lot of contained clones. A contained clone is a clone where

all the tokens of the clone is a part of another, larger clone. The algorithm will give a lot of contained clones because in a case where there is a suffix with an LCP value of 100, the next suffix will have the LCP value of at least 99 and likely matches with the same code clone as the previous suffix, but with an offset of 1. Because of this, for any large code clone, there will be a lot of smaller clones which are completely contained within it, but these clones are also likely to match with another clone which is also contained within the larger clones match. Since the code clones point at mostly the same area, the contained clones are not very useful, and should not be considered. We extend our clone extraction algorithm to account for this, by using the following theorem:

Theorem 4. *The LCP of a suffix at position i is completely contained in the LCP of the previous suffix at position $i-1$ if the LCP value of the suffix at position $i-1$ is greater than the LCP value of the suffix at position i. Meaning:*

$$\text{LCP[ISA[i]]} < \text{LCP[ISA[i} - 1]]$$

Algorithm 7 adds a while-loop to the clone extraction algorithm which skips over all clones with LCP values which should be skipped according to the theorem. Note that this algorithm doesn't disallow contained clones entirely, but any clone which is a shorter version of another clone pointing to the same match, will be skipped. Also note that overlapping clones are allowed, meaning two clones which share tokens, but where neither contains the other.

```
Algorithm CloneExtraction(S, ISA, LCP)
    n ← S_len
    clones ← List
    for i from 0 to n − 1 do
        if ISA[i] = 0 then
            continue
        end

        if LCP[ISA[i]] ≥ THRESHOLD then
            Insert(clones, i) // Adds i to the clone-list
            i ← i + 1
            while i < n and LCP[ISA[i]] < LCP[ISA[i − 1]] do
                i ← i + 1
            end
        end
    end
    return clones
```

Algorithm 7: Extract clones indices in a string S, ignoring contained clones

4.5 Source-mapping

With the clone indices in the fingerprint now computed, we are almost done computing the clones. The final step is to map the clone indices back to the original source code. In order to correctly identify where a clone is located, we need to know which file the clone is located in, and the range of the source code in that file the code clone covers.

To accomplish this, each document in the index needs to store the range of each of its tokens and keep track of which portion of the fingerprint consists of the documents tokens. This is done by storing two integer variables, each storing the start position and end position that the document has in the fingerprint.

Algorithm `SourceMap(`*documents, i*`)`

```
left ← 0
right ← documents_len − 1

while left ≤ right do
    mid ← (left + right)/2
    if documents[mid]_end < i then
        left = mid + 1
    else if documents[mid]_start > i then
        right = mid − 1
    else
        D ← documents[mid]
        range ← D_ranges[i − D_end]
        return (D_uri, range)
    end
end
return
```

Algorithm 8: Get source-map for a position i in the fingerprint

To determine which document a fingerprint position corresponds to, we can perform a binary search on the list of the documents, which is sorted based on the start position of the documents fingerprint. The goal is to find the document D where the fingerprint position i is

$$D_{start} \leq i \leq D_{end}$$

Once the correct document has been found, we simply have to look up the correct range which the document stores. The index of this range is $i - D_{fStart}$.

Algorithm 8 shows this algorithm and how the URI for the document and the range which the token at a position i occupies.

This algorithm only shows how to look up the position of a single token. Since a code clone is a range between two tokens, we have to look up the position of both the fingerprint index i extracted in the previous phase, and the position where the clone ends, which is the fingerprint index $i + \text{LCP}[\text{ISA}[i]]$. The range of the code clone is therefore the combination of the starting range of the first token (at position i) and the ending range of the second token (at position $i + \text{LCP}[\text{ISA}[i]]$). This range is therefore:

$$\text{clone range} = (\text{firstToken.start}, \text{secondToken.end})$$

Aggregating clones

With this algorithm to build the clone objects, the next step is to make sure that matching code clones are collected into buckets of clone classes. Remember that the LCP array only gives us the longest match between two suffixes, but it is of course possible to have more than two clones of the same code snippet. This case happens when multiple consecutive indices in SA are considered to be clones. Since we only look for type-1 clones, the transitivity property holds, meaning that if

$$SA[i] \xrightarrow{clone} SA[i+1] \xrightarrow{clone} SA[i+2]$$

Then clones at position $SA[i]$, $SA[i+1]$ and $SA[i+2]$ are all clones of each other.

Therefore, each code clone object has a list of its matching clones. We will build a clone-map, where the key is an integer which corresponds to which index in the fingerprint that a clone starts at, and the value will be the clone object. For each clone index i which was extracted in the last phase, we get the corresponding match index j (SA[ISA[i] $-$ 1]), and for both i and j we look in the clone-map if there already is a clone at that position, or a new clone object is created and put in the map with that position as the key. Then, in order to aggregate all the clones together, the list of matching clones is unioned between the two clones. In this way, all the previous existing clones of i is added as a clone in j and vice versa.

Chapter 5

Implementation: Incremental detection

The following chapter will present the algorithm which efficiently updates the list of clones, without having to rebuild the different structures from scratch. Given an edit to a file in the project, we will be able to update the document index, fingerprints, suffix array and list of clones faster than the initial detection.

An incremental update is run whenever a document is changed. The document index is signalized of a change either when a file is saved, or on any keystroke. This is configurable by the client. When the document index is changed, an incremental update of the clones is run, and the clone-list is updated.

5.1 Affordable operations

Before looking at the approach, it is useful to determine the time cost associated with different operations. If we can for example afford to iterate over the contents of a single file, that will be useful for our algorithm. Table 5.1 shows which operations are affordable or not for an incremental update.

5.2 Updating the document index

The first step of an incremental update is to update the document index. We will also look at how we can reduce memory usage of the index without a loss in terms of the time complexity of the updates.

As shown in the document interface, each document stores its own content, AST and fingerprint. It is not strictly necessary to store either the content or the AST in memory all the time, as it is likely that only a handful of files are open in the IDE at once. Therefore, in the initial detection, we can free the memory of the file content and AST for each document after the fingerprint has been computed. However, if a file is opened in the IDE, the file can now be changed, so we should facilitate efficient updates for these files only. When a file is opened, the file content

Description	Complexity	Affordable	Explanation
Content of all files	$O(\|\text{code base}\|)$	No	Iterating over the entire code base will be the same complexity as the initial detection, therefore this operation is too slow.
Content of a file	$O(\|\text{file}\|)$	Yes	Iterating over the content of a single file is likely a very small percentage of the entire code base.
Parsing a file	$O(\|\text{file}\|)$	No	While the complexity of parsing a single file is still linear in the size of the file, parsing a large file from scratch can take a significant amount of time in practice.
Incrementally reparsing a file	$O(\|\text{edit}\|)$	Yes	Re-using the AST of a file in order to speed up the parsing of the same file with after an edit is significantly faster than parsing the entire file.
Document index	$O(\|\text{documents}\|)$	Yes	The number of files in a code base is likely many orders of magnitude smaller than the size of the code base itself.
Clones	$O(\|\text{clones}\|)$	Yes	The number of clones in the code base and the area they cover is likely a very small portion of the code base itself. itself.

Table 5.1: Affordable operations for incremental updates

should be read from the disk and updated via the `textDocument/didChange` messages sent from the client. It is also important to keep the AST of the opened file in memory in order to facilitate incremental reparsing of the opened files.

When a file is opened, the LSP client sends a `textDocument/didOpen` message to the server, which finds the relevant document D in the index, and sets the following fields:

$$
\begin{aligned}
D_{open} &= \textbf{True} \\
D_{AST} &= \texttt{Parse}(D) \\
D_{content} &= \texttt{Read}(D_{uri})
\end{aligned}
$$

After the document fields have been set, the document is ready to receive updates. When the LSP client sends a `textDocument/didChange` message, the message consists of the URI of the edited file, the range of the content which has changed, and the content which has potentially been inserted. This range is then used in a tree-sitter incremental reparse of the file content. After this reparse, we have efficiently updated a documents content and AST. After this update, we also set $D_{changed} = \textbf{True}$

5.3 Updating fingerprints

With the updated AST for all documents, we can update the fingerprint of all documents which have been changed. For each document D where $D_{changed} = \textbf{True}$, the fingerprint for D may

have changed. Calculating the fingerprint is the same process as in the initial detection, where we first query the AST for all nodes of a certain type, then for each matched node N, we extract and fingerprint all the tokens which N covers, using the same fingerprint mapping as was used for the initial detection.

An additional change we have to consider when incrementally updating fingerprints is that for a document D, D_{start} and D_{end} which corresponds to the range which D covers in the fingerprint, may have changed. Also, any document D_1 where $D_{1start} > D_{start}$ could also have its range changed. This is solved while updating each documents fingerprint by counting the number of tokens in each document after updating, and setting the appropriate `start` and `end` fields.

TODO: Algorithm for updating document index here

TODO: Figure which displays three documents, with old and new fingerprint

5.4 Extracting edit operations

Now that the fingerprint has been updated, we could build the suffix array from scratch and already see a huge improvement in performance. The major bottleneck of the initial detection is to parse and fingerprint the entire code base. However, the updating of the suffix array can and should also be updated incrementally.

Before we can update the suffix array, we need to know what exactly has changed in the fingerprint. We need to determine what edit operations have happened. Edit operations are either deleting, inserting or substituting a section of the code.

There are two approaches we could take to determine the edit operations. The first is to look at the ranges that the LSP client sends with each `textDocument/didChange` message and determine which tokens in the fingerprint have been affected by this message. However, this approach tightly couples the algorithm to the scenario where we know the exact ranges of each change, which is not very flexible.

The other approach which is more flexible is to determine the changes of the fingerprint via an edit distance algorithm. An edit distance algorithm is an algorithm which calculates the distance between two strings S_1 and S_2. Distance meaning the minimum number of edit operations (insert, delete, substitute) which is required to transform S_1 into S_2. Many of the algorithms which calculates the edit distance also allows computing what the operations are.

The classic algorithm for calculating edit distance operations is attributed to Wagner and Fischer [30]. The input to the algorithm is two strings S_1 and S_2 of length n and m. The output will be the set of operations needed to turn S_1 into S_2. This algorithm is based on dynamic programming where a matrix M is filled from top to bottom and then the operations are inferred from M. Algorithm 9 shows how the edit distance matrix is filled and 5.2 shows an example matrix.

Each index i, j in M contains the edit distance value between the substrings $S_1[0..i]$ and $S_2[0..j]$ The values in M is calculated by determining what is the cheapest operation to do at a certain location to make the substrings equal. This can be determined by looking at the three surrounding indices in M: $M[i-1][j-1]$, $M[i-1][j]$ and $M[i][j-1]$. Each of these indices equate to deleting, inserting or substituting a character in S_1. The algorithm can be boiled down to the

$$\sum_{i=0}^{n} M[i][0] = i$$

$$\sum_{j=0}^{m} M[0][j] = j$$

$$M[i][j] = \begin{cases} M[i-1][j-1] & \text{if } S_1[i] = S_2 j \\ 1 + \min \begin{cases} M[i-1][j-1] \\ M[i][j-1] \\ M[i-1][j] \end{cases} \end{cases}$$

Figure 5.1: Edit distance recurrence

recurrence in figure 5.1

The edit operations can then be inferred from M by backtracking from the bottom-right index, to the top-left, giving us the edit operations in reverse. At each position i, j we choose either of the 3 surrounding indices, the same indices which were used to determine the value originally. Choosing the left index $(i, j-1)$ equates to inserting the character $S_2[j-1]$ at position $i-1$. Choosing the top index $(i-1, j)$ equates to deleting the character $S_1[i-1]$. Choosing the top-left index $(i-1, j-1)$ equates to substituting $S_1[i-1]$ with $S_2[j-1]$. If these characters are already equal, the operation can be ignored.

TODO: Example with DEMOCRAT - REPUBLICAN

TODO: Now collect operations together and explain memory issue + Hirschbergs

5.5 Dynamic suffix arrays

5.6 Storing old clones

```
Algorithm WagnerFischerEditDistance(S1, S2)
    n ← S1len
    m ← S2len
    matrix ← new array[n + 1][m + 1]

    for i from 0 to n do
        matrix[i][0] = i
    end

    for i from 0 to m do
        matrix[0][i] = i
    end

    for i from 1 to n do
        for j from 1 to m do
            if S1[i − 1] = S2[j − 1] then
                matrix[i][j] = matrix[i − 1][j − 1]
            end
            else
                delete ← matrix[i − 1][j]
                insert ← matrix[i][j − 1]
                substitute ← matrix[i − 1][j − 1]

                matrix[i][j] = Min(insert, delete, substitute) + 1
            end
        end
    end
    return matrix
```

Algorithm 9: Fill edit distance matrix using Wagner-Fischer algorithm

		D	E	M	O	C	R	A	T
	0	1	2	3	4	5	6	7	8
R	1	1	2	3	4	5	5	6	7
E	2	2	1	2	3	4	5	6	7
P	3	3	2	2	3	4	5	6	7
U	4	4	3	3	3	4	5	6	7
B	5	5	4	4	4	4	5	6	7
L	6	6	5	5	5	5	5	6	7
I	7	7	6	6	6	6	6	6	7
C	8	8	7	7	7	6	7	7	7
A	9	9	8	8	8	7	7	7	8
N	10	10	9	9	9	8	8	8	8

Table 5.2: Edit distance matrix

Chapter 6

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 [27] 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.

6.1 Big-O analysis of incremental detection

6.2 Performance comparison

6.3 Comparison with iClones

6.4 Verifying clones

Chapter 7

Discussion

7.1 Speed of incremental updates vs linear-time SACA

7.2 Comparison with iClones

7.3 Chosen clones

7.4 Usability while programming

Chapter 8

Conclusion and future work

8.1 Future work

Compressing data structures

Optimal edit operations

Delaying LCP array updates

Refactoring of clones

8.2 Conclusion

Bibliography

[1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. "Replacing suffix trees with enhanced suffix arrays". In: *Journal of Discrete Algorithms* 2.1 (2004). The 9th International Symposium on String Processing and Information Retrieval, pp. 53–86. ISSN: 1570-8667. DOI: https://doi.org/10.1016/S1570-8667(03)00065-0. URL: https://www.sciencedirect.com/science/article/pii/S1570866703000650.

[2] 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.

[3] 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.

[4] 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.

[5] Ira Baxter et al. "Clone Detection Using Abstract Syntax Trees." In: vol. 368-377. Jan. 1998, pp. 368–377. DOI: 10.1109/ICSM.1998.738528.

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

[7] Michael Burrows and David J. Wheeler. "A Block-sorting Lossless Data Compression Algorithm". In: 1994.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] G. Jacobson. "Space-efficient static trees and graphs". In: *30th Annual Symposium on Foundations of Computer Science*. 1989, pp. 549–554. DOI: 10.1109/SFCS.1989.63533.

[16] Lingxiao Jiang et al. "DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones". In: *29th International Conference on Software Engineering (ICSE'07)*. 2007, pp. 96–105. DOI: 10.1109/ICSE.2007.30.

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

[18] 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.

[19] 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.

[20] Microsoft. *Language Server Protocol*. https://microsoft.github.io/language-server-protocol/. Accessed: 2023-02-17.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] M. Salson et al. "Dynamic extended suffix arrays". In: *Journal of Discrete Algorithms* 8.2 (2010). Selected papers from the 3rd Algorithms and Complexity in Durham Workshop ACiD 2007, pp. 241–257. ISSN: 1570-8667. DOI: https://doi.org/10.1016/j.jda.2009.02.007. URL: https://www.sciencedirect.com/science/article/pii/S1570866709000343.

[27] 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.

[28] 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.

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

[30] Robert A. Wagner and Michael J. Fischer. "The String-to-String Correction Problem". In: *J. ACM* 21.1 (Jan. 1974), pp. 168–173. ISSN: 0004-5411. DOI: 10.1145/321796.321811. URL: https://doi.org/10.1145/321796.321811.

[31] 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.

[32] 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.

[33] 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.