UNIVERSITY
OF OSLO

Master's thesis

Incremental clone detection for IDEs using dynamic suffix arrays

CCDetect-LSP: Detecting duplicate code incrementally in IDEs

Jakob Konrad Hansen

60 ECTS study points

Department of Informatics
Faculty of Mathematics and Natural Sciences

Spring 2023

Jakob Konrad Hansen

Incremental clone detection for IDEs using dynamic suffix arrays

CCDetect-LSP: Detecting duplicate code incrementally in IDEs

Abstract

TODO: Write abstract

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 [12, p. 9]. Refactoring source code is often performed in order to eliminate instances of bad code quality, 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, code which is more or less copied to different locations in a code base, 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 [18, 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 IDEs.

Our proposed solution and tool addresses this issue by introducing a new algorithm that is capable of detecting and updating existing code clones as source code changes, which is faster than redoing the analysis from scratch. CCDetect-LSP, 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 Our contribution

CCDetect-LSP provides clone detection capabilities in a real-time IDE environment. The goal of this tool 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 code base, 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 algorithms 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.

1.3 Structure

Chapter 2 provides 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 multiple 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 and research opportunities.

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" [11, 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 [21, p. 29]. Whenever duplicated code needs to changed, it will require changes in multiple locations, meaning that metrics such as maintainability, analyzability and changeability are negatively affected.

These metrics are affected negatively by duplicated 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 anti-patterns. Anti-patterns are bad design decisions for software which can lead to 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" [12, p. 66] anti-pattern. This anti-pattern 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", 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.*

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

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

    print(i);
}
```

Figure 2.1: Type-1 clone pair

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 clone pair. Similarity is determined by some metric such as number of equal lines of code.*

Definition 3 (Clone class)**.** *A clone class 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 and c are both clones of b, but not necessarily similar enough to be clones of each other, 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 classes.

Code clones are generally classified into four types [18]. 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.

Type-1 clones are syntactically identical. The only differences for a pair of type-1 clones are elements without meaning, like comments and white-space. This is code which is generally ignored when source code is lexed and parsed. Figure 2.1 shows an example of a type-1 clone pair where only a comment and white-space is added to the snippet on the right.

Type-2 clones are structurally identical. Possible differences include changes to 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 [43]. This type of clone is relevant to consider in refactoring scenarios when merging code clones as they can be relatively simple to parameterize in order to merge two clones with differing literals for example. The original locations of clones can then be replaced with a call to the merged code with different parameters. Figure 2.2 shows an example of a type-2 clone pair where the numbers in the for loop initialization and condition could be parameterized to

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

```
for (int j = 5; j < 20; j++)
    {
    print(j);
}
```

Figure 2.2: Type-2 clone pair

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

```
for (int i = 0; i < 10; i++) {
    print(i);
    print(i*2);
}
```

Figure 2.3: Type-3 clone pair

correctly merge the two clones.

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 [18, p. 6]. The granularity for this difference could for example be based on the number of differing tokens or lines. Detecting this type of clone is generally harder and takes more computation than type-1 and type-2 clones. Figure 2.3 shows two code snippets where the code snippet on the right has an added statement.

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 [42]. Figure 2.4 shows two code snippets which have no clear syntactic or structural similarity, but is functionally equal.

Type-1 clones are often referred to as "exact" clones, while Type-2 and Type-3 clones are referred to as "near-miss" clones [43, 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 [18]. 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 gran-

```
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);
```

Figure 2.4: Type-4 clone pair

ularity 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 [32]:

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 [43]. 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 [20].

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 [43].

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 [14]. 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. [28] 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. [17] 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 [30] 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 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" [31, 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 [37, 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 [43]. It was later used in the tool *SimEclipse* [37] 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

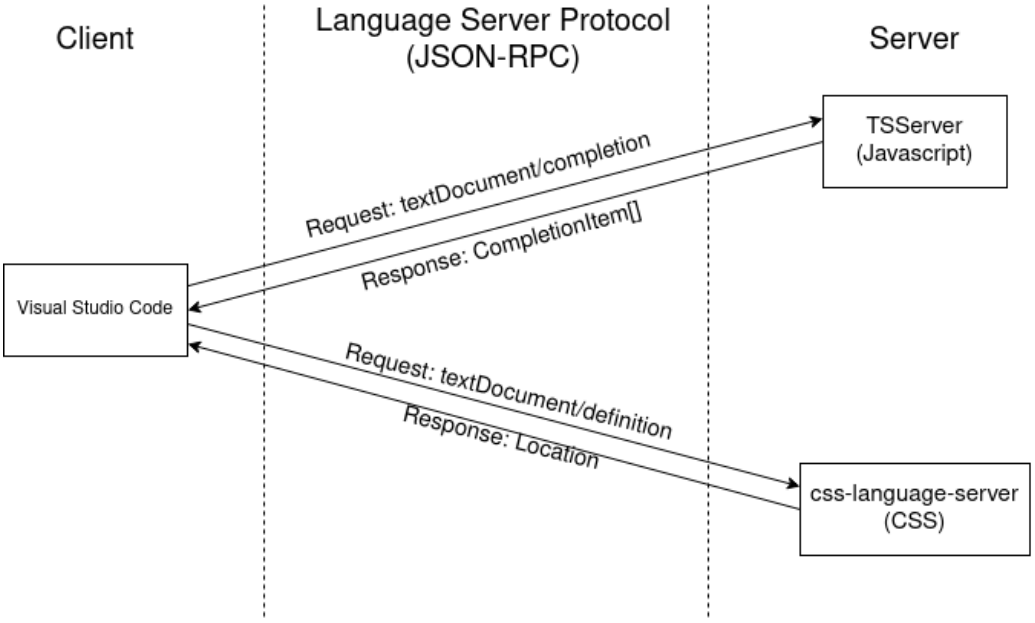


Figure 2.5: Example client-server interaction using LSP

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 [34], one can assume that the suffix array is not dynamically updated. [23].

- 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.

2.6 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 [27].

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 2.5 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.

2.7 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

While writing code, programmers usually only edit small regions of code 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.

In our detection algorithm we will need to be able to parse our source code into an AST representation in order to select specific syntactic regions and extract the tokens. Therefore, we need a parsing algorithm, and when source code is edited we need an incremental parsing algorithm to more efficiently parse the source code when only a small region has been changed.

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.

`Tree-sitter` is a parser generator tool which specializes in incremental parsing. 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.

The incremental reparsing in Tree-sitter is performed using Wagner's algorithm [41].

TODO: Explain Wagner's algorithm with figure and example

Suffix trees

A classic algorithm [43, 14] 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. 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 2.6 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

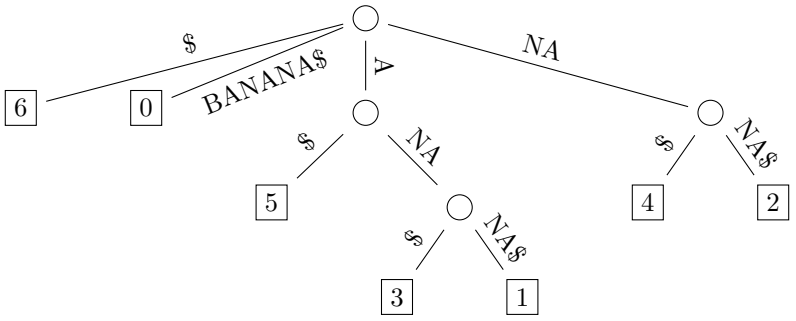


Figure 2.6: Suffix tree for T = BANANA$

suffixes before it needs to, lowering the complexity [39].

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.*

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 [43, 14] 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 2.6, 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

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$

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 describes how many common characters 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 [22], 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 [29].

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 on bitset)**.** *A rank query $rank_1(i)$ on a bitset B, computes 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 on bitset)**.** *A select query $select_1(i)$ on a bitset B, computes 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 [19].

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

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 insertions/deletions 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. Keeping the bitset balanced is done by splitting a leaf-node bitset to two nodes when the leaf-node bitset has reached a certain size (such as $2 \times \log(n)$) and rebalancing the tree after the split. Similarly, when considering deletions, two leaf-node bitsets can be merged to their parent node when their combined number of bits has decreased to $\log(n)$.

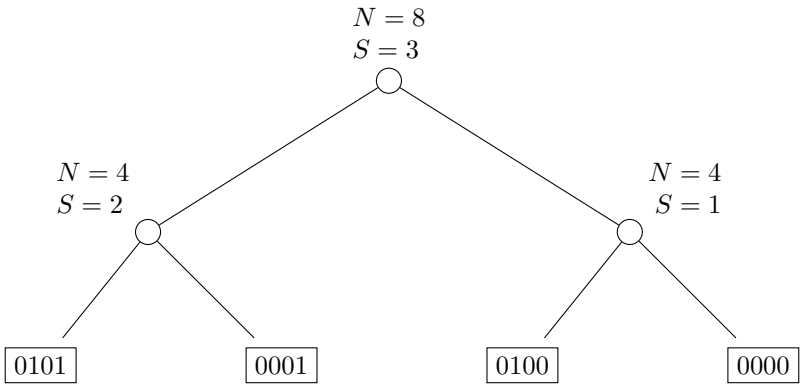


Figure 2.7: Dynamic bitset

Figure 2.7 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

Wavelet tree and wavelet matrix

We can extend the notion of rank and select queries to strings as well.

Definition 10 (Rank query on string)**.** *A rank query $rank_c(i)$ on a string S computes the number of occurrences of c in S up to, but not including position i.*

Definition 11 (Select query on string)**.** *A select query $select_a(i)$ on a string S, computes the position of the ith occurrence of a in S.*

The classic data structure to efficiently compute *rank* and *select* queries on strings is the **wavelet tree** [15]. The data structure is a binary tree where every node consists of a bitset, and each level of the tree consists of the bits of a single position in each character of the string. Each bitset in the wavelet tree can for example be implemented as a dynamic bitset to allow insertions/deletions on a wavelet tree.

Figure 2.8 shows the wavelet tree for the string `abbbefcagdd`. The wavelet tree can perform *access*, *rank* and *select* queries by traversing the tree from root to leaf.

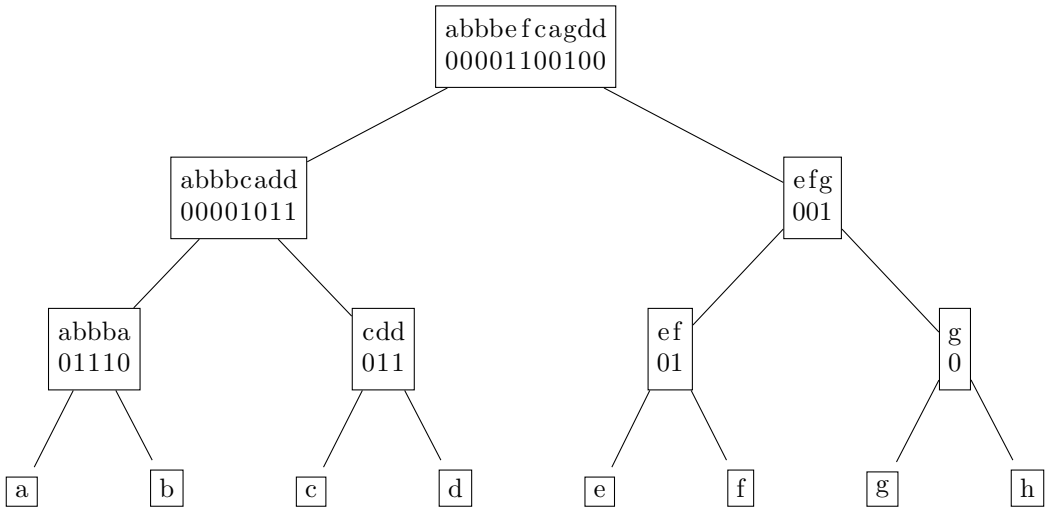


Figure 2.8: Wavelet tree for S = abbbefcagdd

a	b	b	b	e	f	c	a	g	d	d
0	0	0	0	1	1	0	0	1	0	0
0	1	1	1	1	0	1	1	0	1	1
0	1	1	1	0	0	1	1	0	1	0

Table 2.2: Levelwise wavelet tree

access(i) gives the bitstring of the character at a position i by first accessing position i in the root bitset, if the bit is a 0, we traverse to the left, if it is a 1 we traverse to the right. Before traversing to a child, we compute $rank_0(i)$ or $rank_1(i)$, depending on what bit is at position i, the resulting rank is the next position to consider in the subtree. This is done recursively until a leaf-node is found, and the bits which were examined at each node is the bitstring of the character at position i.

$rank_c(i)$ traverses the tree similarly to *access*(i), but we use the bits of c to guide us to the correct subtree. When a leaf-node is reached, the value of i is returned, since when a leaf is reached, i will be pointing to the correct rank in the fictitious bitset of only a characters

The wavelet tree can be traversed in $O(\log \sigma)$ time where σ is the size of the alphabet. However, the *access*, *rank* and *select* operations also depend on how fast the bitsets can perform *access*, *rank* and *select* operations. If the bitsets are implemented using for example Jacobson's rank [19], the bitsets have a $O(1)$ *access*, *rank* and *select* time, but if we require dynamic bitsets for insertion/deletions, the time complexity increases to $O(\log n \log \sigma)$ because a bitset operation which takes $O(\log n)$ time is required in each level of the tree.

The wavelet tree can also be implemented without pointers, known as a levelwise or pointerless wavelet tree[26]. Table 2.2 shows the levelwise wavelet tree, which can be traversed level by level, but determining which interval of the bitset the node you are in occupies requires two calls to *rank*. The details of the levelwise wavelet trees are out of scope for this thesis, but leads us into the next wavelet data structure.

a	b	b	b	e	f	c	a	g	d	d
0	0	0	0	1	1	0	0	1	0	0
0	0	0	0	1	0	1	1	0	0	1
0	1	1	1	0	0	1	0	1	1	0

Table 2.3: Wavelet matrix for S = abbbefcagdd

Algorithm `access(`*WM, i*`)`

```
bits ← empty list
l ← 0
while l < Len(WM) do
    Add(bits, access(WM[l], i))
    if access(WM[l], i) = 1 then
        i ← z_l + rank_1(WM[l], i)
    else
        i ← rank_0(WM[l], i)
    end
    l ← l + 1
end
return bits
```

Algorithm 2: Wavelet matrix access

Algorithm $\texttt{rank}_c$`(`*WM, i*`)`

```
l ← 0
p ← 0
while l < Len(WM) do
    if access(WM[l], i) = 1 then
        i ← z_l + rank_1(WM[l], i)
        p ← z_l + rank_1(WM[l], p)
    else
        i ← rank_0(WM[l], i)
        p ← rank_0(WM[l], p)
    end
    l ← l + 1
end
return i − p
```

Algorithm 3: Wavelet matrix rank

The **wavelet matrix** is a relatively recent improvement on the wavelet tree, which has been shown to be more efficient for *access*, *rank* and *select* queries, as well as less memory intensive for larger alphabets [9]. As the name suggests, the wavelet matrix is a matrix of bitsets, instead of a tree of bitsets. Similarly to the levelwise wavelet tree, for a string S of size n, each level of the matrix consists of a bitset of size n. The difference between a levelwise wavelet tree and a wavelet matrix is that the assumption that the "children" of an interval $[x, y]$ needs to occupy exactly $[x, y]$ is no longer true. We can instead put all 0 bits to the left in the next level, and all 1 bits to the right. For each level we store an integer z_l, which holds the number of zero bits in level l.

For *access* operations, we traverse each level similarly to a levelwise wavelet tree, but instead of traversing a smaller and smaller interval in each level, we simply look at the current bit at position i, if it is a zero, the bit to examine in the next level is $rank_0(i)$, if the bit is a 1, the bit to examine in the next level is $z_l + rank_1(i)$. $rank_c(i)$ operation is carried out similarly, but we also keep track of a preceding position to the final i, which we subtract from i, to count only the number of proceeding number of a characters. Table 2.3 shows an example wavelet matrix, and algorithm 2 and 3 shows how *access* and *rank* operations can be carried out for a wavelet matrix. The time complexities for *access*, *rank* and *select* operations is the same as for wavelet trees.

The wavelet matrix is constructed level by level. In level i, the bit at position i in each character is inserted. Before constructing the next level, each character is sorted by the current bit, so that all the characters with 0 bits at position i occupies the left side of level $i + 1$, and vice versa for 1 bits.

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 function

Table 2.4: S = BANANA$, BWT = ANNB$AA

Burrows-Wheeler transform

The Burrows-Wheeler transform (BWT) is a transform on strings, often performed to improve compression and searching [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 2.4 shows the BWT for the string S = BANANA$.

Definition 12. *Cyclic-shift*

The cyclic-shift of order i for a string S is the cyclic-shift of S such that all characters in S are rotated i characters to the left. The cyclic-shift of order i is denoted CS_i

Definition 13. *Burrows-Wheeler transform*

The Burrows-Wheeler transform of a string S is a transformation on S where the cyclic-shifts of S are sorted, and the final character in each shift is concatenated.

There is a direct correlation between the suffix array of S and the BWT of S. Table 2.4 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 4 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.

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 S. 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).

S can be calculated from the BWT as long as there is a unique terminating character ($), or the

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] = S[pos]
end
return BWT
```

Algorithm 4: Calculating the BWT of a string T from its suffix array

position of the final character is stored. S is computed backwards by starting at the final character ($S[n-1]$) 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 $S[n-2]$. 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 S, 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 S 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 function (LF function). Calculating the LF function can be done efficiently by using a rank/select data structure such as a wavelet tree [15] or wavelet matrix [9] 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 function will also be useful in the detection algorithm when dynamically updating the suffix array.

Chapter 3

Implementation: LSP server architecture

CCDetect-LSP is integrated into IDEs via the Language server protocol. The tool starts up as an LSP server, which an IDE client can connect to and send/receive messages from. The goal of the LSP client-server interaction is to give users of the tool an overview of all clones as they appear. The LSP server is implemented in Java with the LSP4J library which provides an abstraction layer on top of the protocol which is easier to work with programmatically.

The LSP server shows error-messages (diagnostics) which indicate which section of code a clone covers, they also provide information about the matching clones and code-actions to navigate between them. Whenever source code is edited, the LSP server receives a message containing information about what has changed. The LSP server then generates a new list of clones, and displays those to the client instead.

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 one of the matches 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” refac-

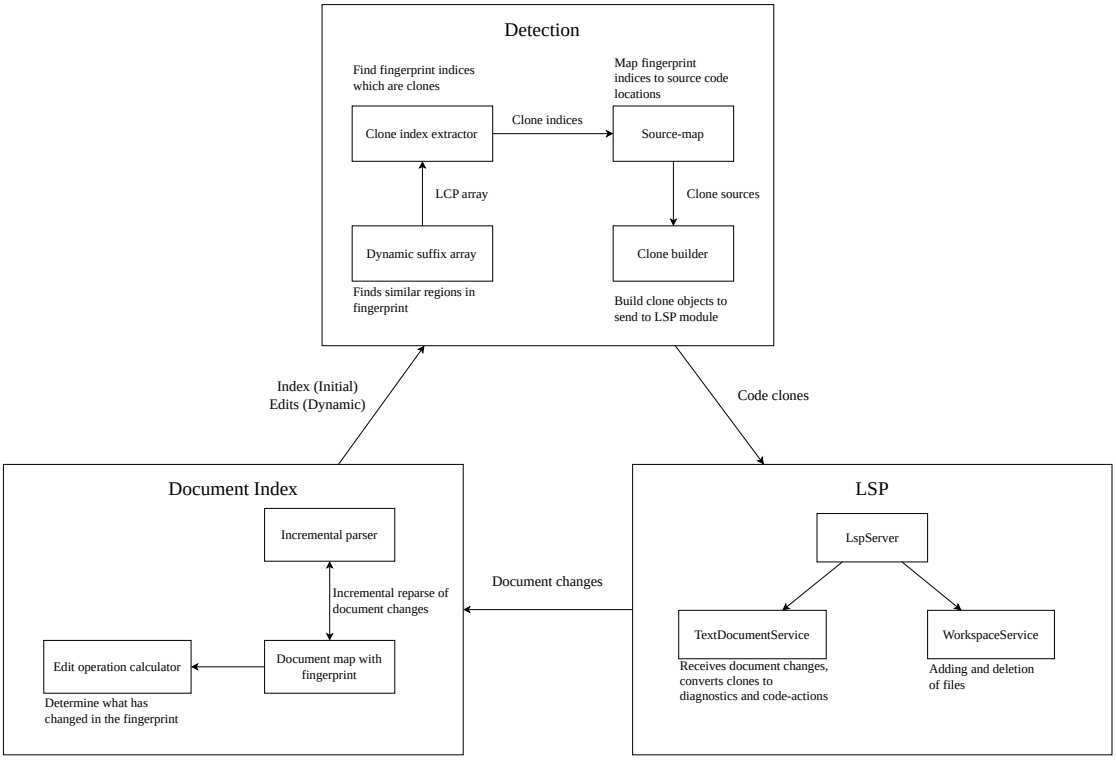


Figure 3.1: Tool architecture

toring. The programmer performs the necessary refactorings, and will see that all the previously detected clones are gone when the LSP server updates.

Figure 3.1 shows the architecture of the tool. The LSP module communicates with the IDE and delegates the work of handling documents to the document index. Detection of clones is delegated to the detection module. The LSP module receives a list of clones from the detection module which is then converted to LSP diagnostics and code-actions which is finally sent to the client.

3.1 Document index

Upon starting, the LSP server requires indexing of the project for conducting analysis. This involves creating an index and inserting all the relevant documents in the code base. 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 record for our documents:

```
record 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 generally 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 `open` 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

When detection is finished and the list of clones is ready, the LSP module will convert clones into diagnostics and code-actions which can be interacted with from the client. For diagnostics, each clone object is converted into a diagnostic which has a range and some information about the clone. The diagnostic displays an error in the client which also has information for each matching clone. This is achieved this by attaching `DiagnosticRelatedInformation` to a diagnostic which is defined in the protocol. When all the code clones have been converted to diagnostics, the server sends a `textDocument/publishDiagnostics` which sends all the diagnostics in JSON-RPC format. For code-actions, the process is similar, for each clone pair, we create a code-action which navigates from one to the other, using the `window/showDocument` message. These code-

actions are similarly sent to the client via the `textDocument/codeAction` message, but this message is only invoked when the client specifically requests code actions at a certain location.

Figure 3.2 shows how clones are displayed in the VSCode IDE. Code clones are displayed inline in the file as an "Information" diagnostic, and when hovered over, the client shows the `DiagnosticRelatedInformation` information of the diagnostic where all the matching clones are displayed and can be clicked to navigate to the corresponding match. For a client which may not implement the `DiagnosticRelatedInformation` message (which was added in LSP version 3.16), invoking the code-action to navigate to the matching clone is another option.

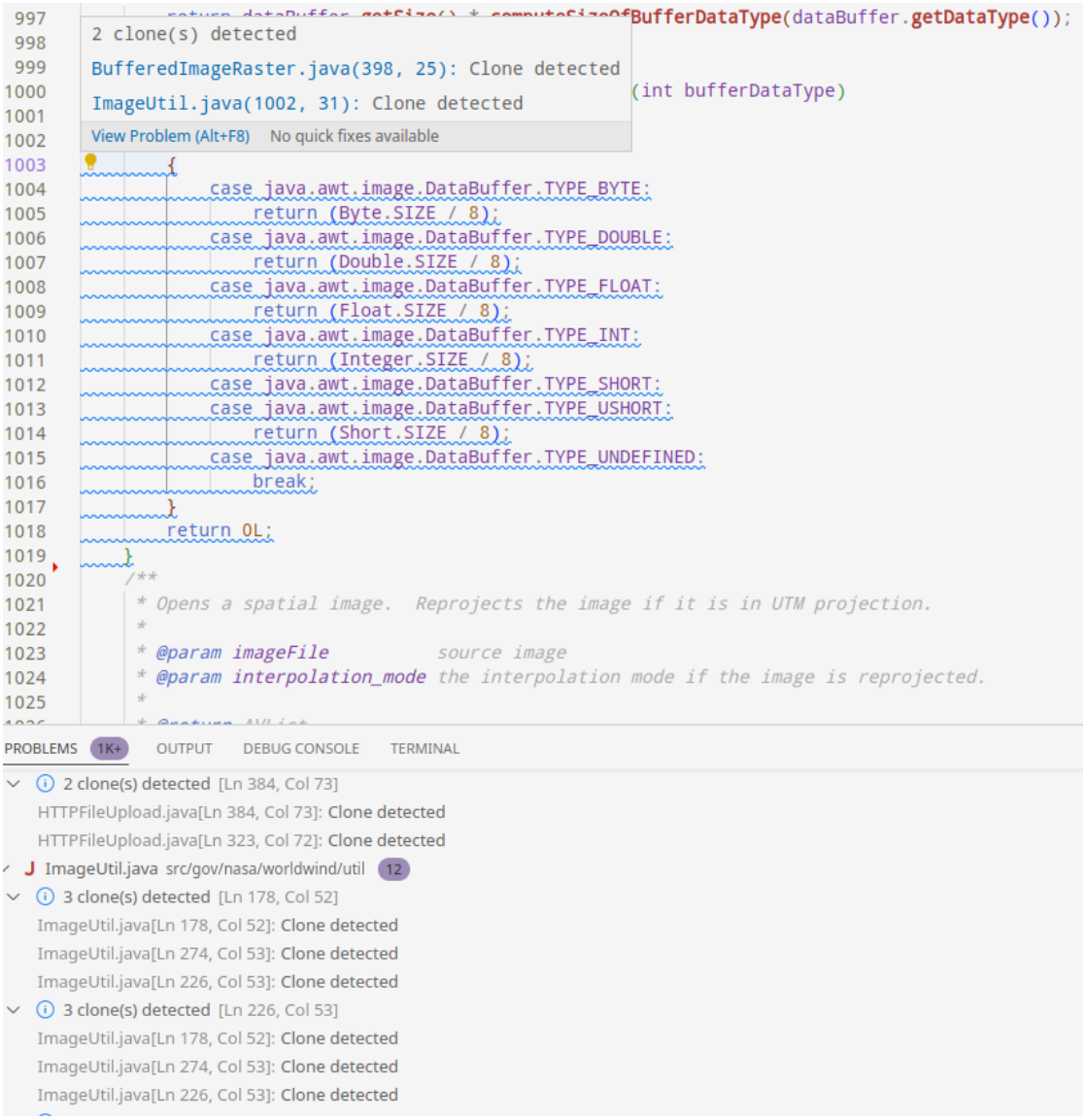


Figure 3.2: Code clones displayed in the VSCode IDE

Chapter 4

Implementation: Initial detection

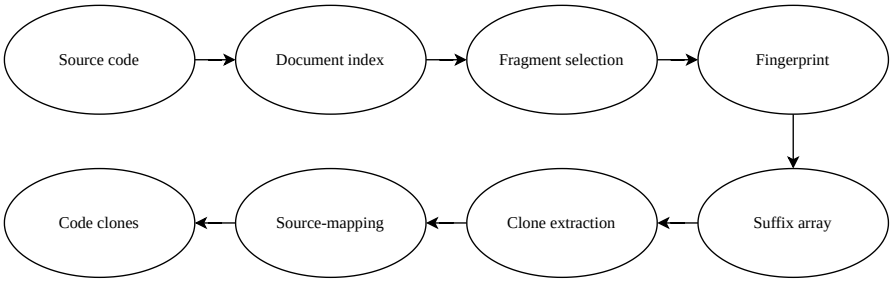


Figure 4.1: Phases of the detection algorithm

This chapter discusses the detection module of CCDetect-LSP and how the initial detection algorithm finds code clones. 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. Figure 4.1 shows all the phases of the detection algorithm and in each section a figure will be shown to illustrate which part of the pipeline is being discussed. The previous chapter has already detailed the first two phases, how a document index is built for each file of the source code.

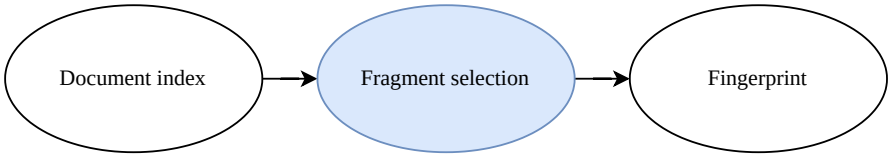
The algorithm detects syntactical type-1 code clones, based on a token-threshold. Clones detected are therefore snippets of source-code which has at least N equal tokens, where N is a configurable parameter. There are also two types of clones which are filtered:

- Clones which are completely contained within another larger clone are filtered.
- Clones cannot extend past the end of a 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 long 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 involves extracting 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, a parser generator tool is used, which can generate the code for parsing any programming language given a grammar. We use Tree-sitter, a parser generator with incremental parsing and error-recovery capabilities, as well as a query language for finding any type of node in the AST [6]. Tree-sitter queries are flexible queries to extract specific nodes or subtrees. 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:

$$\text{(method\_declaration @method)} \tag{4.1}$$

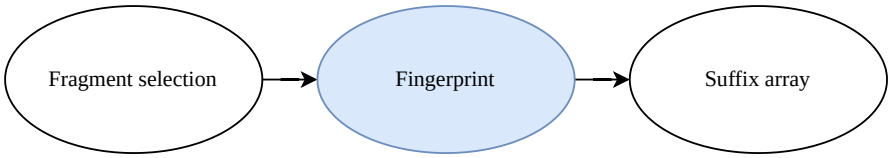
This Tree-sitter query selects the node with type `method_declaration` and "captures" it with the `@method` name, so that it can be further processed by the program. Readers interested in the details of the query system are referred to the Tree-sitter documentation [6].

Using a tree-sitter query, the algorithm parses the program into an AST and queries the tree for a list of all nodes which matches the query. For each node, all the tokens in the subtree of the node are extracted and sent to the next phase.

TODO: Algorithm?

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 [13, p. 366].

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 underlined

Figure 4.2: Example fingerprint of Java source-code

The next phase of the algorithm is to transform the extracted tokens 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.

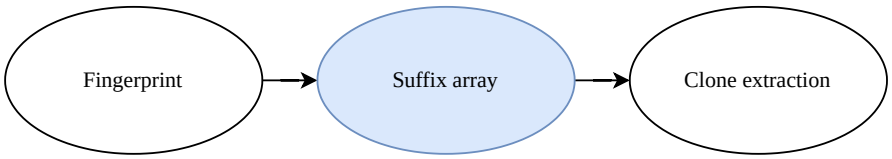
Since the algorithm that is used for matching is based on a suffix array, the representation should be in a format similar to a string, which is the standard input to a suffix array construction algorithm. However, it is not strictly necessary to use strings. The essential property of the input array is that we have a sizeable alphabet where each element is comparable. We will use an array of integers instead of a string as it has a large alphabet in most programming languages.

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 the bit string of an integer. Each unique token value is mapped to unique integer values. Figure 4.2 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 the relevant document object.

B	A	N	A	N	A	$
L	S	L	S	L	L	S
0	1	2	3	4	5	6

Table 4.1: Suffix types of S = BANANA$, LMS characters underlined

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 repeating sequences. Recall that the suffix array of a string sorts all the suffixes of the string, as discussed in section 2.7. This makes it simple to find similar regions of text, since similar suffixes will be next to each other in the suffix array. All the fingerprints which are stored in each document object will now be 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 [29] 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 assumed to have a string as its input, as this is most common for suffix arrays, and working with strings can be more clear to the reader when considering 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 suffix array and inducing how to sort the suffixes of the string S from a smaller string, S_1. S_1 consists of the "building blocks" of S and will make it simple to compute the rest of the suffix array of S. 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 14 (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 in the same string cannot be lexicographically equal, therefore all cases are handled by this definition. Determining the type of each suffix can be done in $O(n)$ time 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 true because 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

$$\text{types[n]} = S$$

$$\sum_{i=0}^{n-1} \text{types[i]} = \begin{cases} S & \text{if S[i]} < \text{S[i} + 1] \\ L & \text{if S[i]} > \text{S[i} + 1] \\ \text{types[i} + 1] & \text{if S[i]} = \text{S[i} + 1] \end{cases}$$

Figure 4.3: Suffix type recurrence

Buckets	$	A	B	N
Initial	{ -1 }	{ -1, -1, -1 }	{ -1 }	{ -1, -1 }
Slot LMS	{ 6 }	{ -1, 3, 1 }	{ -1 }	{ -1, -1 }
Slot L-type	{ 6 }	{ 5, 3, 1 }	{ 0 }	{ 4, 2 }
Slot S-type	{ 6 }	{ 5, 3, 1 }	{ 0 }	{ 4, 2 }

Mapping	$	0
	ANA$	1
	ANA	2

S_1	210
SA of S_1	[2, 1, 0]
Sorted LMS-substrings	[6, 3, 1]

Table 4.2: Building S_1 and sorting LMS-substrings of S = BANANA$

the first character. If the first character is equal, we have effectively transformed the problem to finding the type of the next suffix, since we are now comparing the second character of the current suffix, with the second character of the second suffix. Since we have already computed the type of the next suffix, we can reuse the value. Figure 4.3 shows a recurrence which determines the type of each suffix in a string in $O(n)$ time and 4.1 shows an example.

Definition 15 (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 an LMS character and the first character is never an LMS character.*

Definition 16 (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 characters 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 is mostly lexicographically decreasing or increasing, which is easier to sort. Table 4.2 shows that in the string BANANA$ there are 3 S-type suffixes and all of them form LMS-substrings (AN, ANA, $). Using this notion, we can sort all suffixes recursively using 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.*

We can construct a smaller string S_1 by first sorting the LMS-substrings. Sorting LMS-substrings can be done by first bucket-sorting each LMS-substring by its first character. The buckets are represented by arrays and each LMS-substring is inserted at the end of the correct bucket. Afterwards, L-type suffixes are bucketed by iterating over the buckets, and for each suffix in the bucket, insert the suffix to the left, if it is L-type. Meaning that if we encounter the suffix at position 6, the suffix at position 5 is bucketed if it is L-type. Finally, S-type suffixes are bucketed similarly in the same fashion, but the buckets are scanned from right-to-left and suffixes are inserted at the end of the bucket. This final step could possibly change the ordering of the LMS-substrings.

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. S_1 is now built by mapping each LMS-substring to its unique value, and concatenating them in the original ordering. S_1 is now a smaller 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. Nong et al. 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 [29].

The base-case of the recursion is when the suffix array can be computed simply by bucketing each suffix, which happens when every suffix of the string starts with a unique character.

Table 4.2 shows how the LMS-substrings are bucketed and how S_1 is constructed. We see that the final buckets are actually equal to the final SA which we are trying to compute. This is because the LMS-substrings are already sorted in reverse order, which is not true for any arbitrary input. S_1 consists of only unique characters, therefore the suffix array of S_1 is computed by simply bucketing each suffix and returning the array.

When the recursive call returns, the SA of S_1 is used to determine the order which the LMS-substrings should be slotted into the bigger SA. By scanning the smaller SA and mapping those indices back to the indices of the original LMS-substrings, we get a sorted ordering of the LMS-substrings. We then scan the sorted LMS-substrings from right-to-left and slot each LMS-substring at the end of its bucket, the rest of the L-type and S-type suffixes will be slotted correctly afterwards. For BANANA\$ this will be the exact same process as in table 4.2, since the LMS-substrings were already inserted in reverse ordering.

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:

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

Building ISA and LCP arrays

Computing the ISA after constructing the SA is simple. Since the ISA is simply the inverse of SA, it can be constructed in linear time with a single loop as seen in Algorithm 5.

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 5: Compute ISA from SA

Computing the LCP in linear time is more complicated and requires some insight 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 array. The algorithm to compute the LCP is shown in Algorithm 6. The intuition for this algorithm is that if a suffix at position i 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 6: Compute LCP from input string S, SA, and ISA

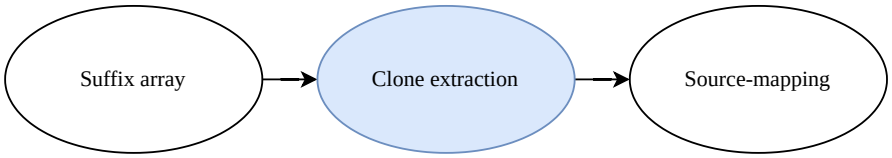
Index	Suffix						Minimum LCP
1	A	N	A	N	A	$	0
3	A	N	A	$			0
2	N	A	N	A	$		2
4	N	A	$				2

Table 4.3: Minimum common LCP values between suffixes for S = BANANA$

```
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 7: Extract clones indices in a string S

4.4 Clone extraction



With the extended suffix array computed, 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 be mapped back to the original source code in the next phase.

A straightforward solution is to extract every suffix which has an LCP value which is greater than the token threshold. The algorithm is a loop over S, using ISA to find the corresponding LCP value. This finds the clone indices, as shown in Algorithm 7.

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, for example in the 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 token. For any large code clone, there will therefore be many 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* $\text{LCP}[\text{ISA}[i]] < \text{LCP}[\text{ISA}[i-1]]$.

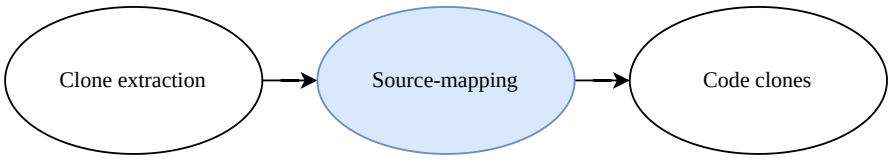
```
Algorithm CloneExtraction(S, ISA, LCP)
    n ← S.len
    clones ← empty list
    for i from 0 to n − 1 do
        if ISA[i] = 0 then
            continue
        end

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

            while i + 1 < n and LCP[ISA[i + 1]] < LCP[ISA[i]] do
                i ← i + 1
            end
        end
    end
    return clones
```

Algorithm 8: Extract clones indices in a string S, ignoring contained clones

Algorithm 8 adds a while-loop to the clone extraction algorithm which skips over suffixes which are contained 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, meaning two clones which share tokens, but where neither contains the other, will not be skipped.

Finally, we have a list of indices in the fingerprint which is considered to be code clones.

4.5 Source-mapping



With the clone indices in the fingerprint computed, we are almost finished. 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, the range of the source code in that file the code clone covers, and the other matching code clones.

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.

To determine which document a fingerprint position corresponds to, we can perform a binary

```
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
```

Algorithm 9: Get source-map for a position i in the fingerprint

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 $i - D.start$.

Algorithm 9 shows this algorithm which outputs the URI for the document and the source code range of the token at position i.

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 at index i extracted in the previous phase, and the position where the clone ends, which is 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 $i + \text{LCP}[\text{ISA}[i]]$):

Aggregating clones

With this algorithm to get the clone locations, 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 naturally possible to have more than two clones of the same code snippet. This case happens when multiple consecutive indices in the SA are considered to be clones. Since we only look for type-1 clones, the transitivity property holds, meaning that if

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

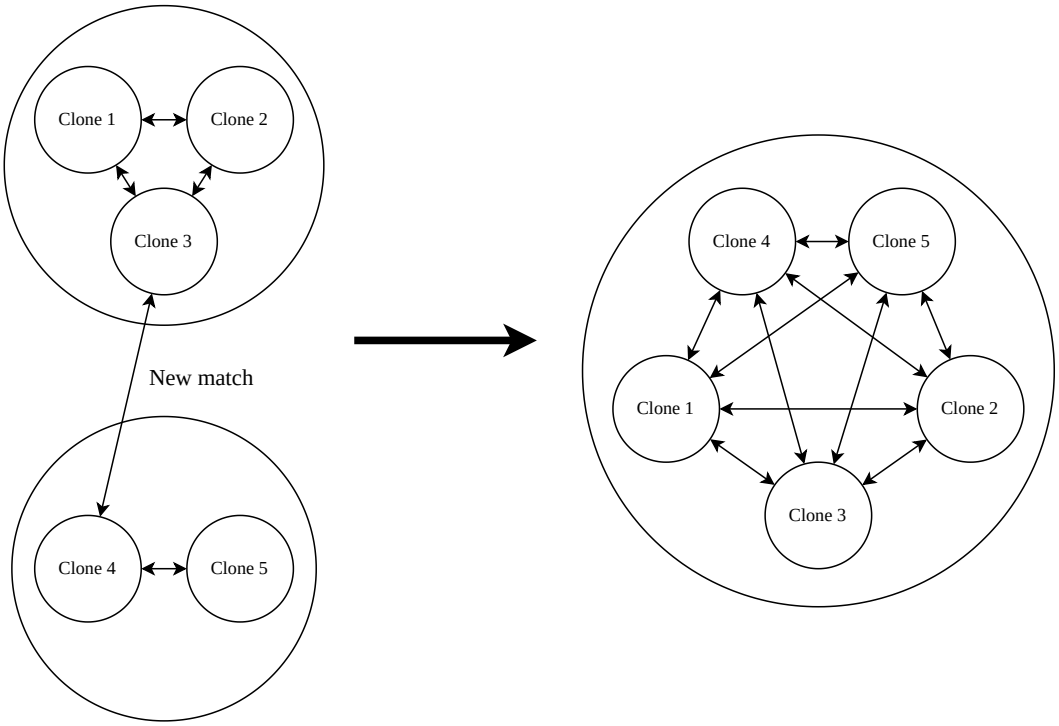


Figure 4.4: Clone class aggregation when a new match is found

then clones at position $SA[i]$, $SA[i+1]$ and $SA[i+2]$ are all clones of each other. This should be achieved by making sure that every new clone-pair discovered adds previously detected clones to their clone sets, and previously discovered clones also add new clones to theirs. Figure 4.4 shows how a new match is found in two previously disjoint clone classes, and the resulting aggregated clone class.

This is achieved in algorithm 10 where we build a clone-map, where the key is the index that a clone starts at in the fingerprint. 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. Then, in order to aggregate the previously discovered clones and the new clone together, the set 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. Every clone in the two clone classes will then receive the same set of code clones. The `UnionCloneClass` function unions all the clone sets in both clone classes and then adds a match from all clones in one clone class to all clones in the other.

Finally, we have a list of every code clone in the code base, which is then sent to the LSP module of the tool, which handles the displaying of code clones to the client as shown in chapter 3.

```
Algorithm GetCloneMap(index, cloneIndices)
    n ← cloneIndices.len
    cloneMap ← Empty map with type: int → CodeClone

    for i from 0 to n − 1 do
        firstIndex ← cloneIndices[i]
        secondIndex ← SA[ISA[firstIndex] − 1]
        size ← LCP[ISA[firstIndex]] − 1

        // Use SourceMap function to get ranges and documents of clones
        firstRange ← Get source between firstIndex and (firstIndex + size)
        secondRange ← Get source between secondIndex and (secondIndex + size)

        firstDocument ← firstRange.document
        secondDocument ← secondRange.document

        // Build clone objects if they don't exist
        if firstIndex not in cloneMap then
            firstClone ← new CodeClone(firstDocument.uri, firstRange)
            Put(cloneMap, firstIndex, firstClone) // Put clone with firstIndex as key
        end
        if secondIndex not in cloneMap then
            secondClone ← new CodeClone(secondDocument.uri, secondRange)
            Put(cloneMap, i, secondClone) // Put clone with secondIndex as key
        end

        // Union clone classes
        UnionCloneClass(Get(cloneMap, firstIndex), Get(cloneMap, secondIndex))
    end
    return cloneMap
```

Algorithm 10: Build clone-map given the clone indices in the fingerprint

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. We can for example afford to iterate over the contents of a single file, which will be useful for our algorithm.

We cannot afford to iterate over the entire code base, meaning the contents of all files in the code base. The initial detection takes linear time in the size of the code base, therefore, iterating over the entire code base will likely make the algorithm as slow as the initial detection.

As mentioned, we can afford to iterate over the contents of a single file. Iterating over the contents of a file with up to a few thousands lines is not a very expensive operation to perform and will not take a significant amount of time. The whole string of the file contents will be needed for Tree-sitter to incrementally re-parse the file.

However, parsing an entire file can be too expensive. While the runtime of parsing a single file is still linear in the size of file content, parsing a large file from scratch can take a significant amount of time in practice, and for small code bases with large files, parsing will take a significant portion of the total runtime of a dynamic update. This is why incremental re-parsing with Tree-sitter is

used, which lowers the runtime closer to $O(|\text{edit}|)$, rather than $O(|\text{file}|)$

We can also afford two more useful operations, iterating over the documents in the index (not their contents), and iterating over the clone objects. We can afford to do these operations because the number of documents and the number of clones is likely multiple magnitudes smaller than the size of the entire code base. Iterating over the documents will be useful when we are updating the document index, and iterating over the clones is necessary to do when the clones are mapped back to their original source code locations.

Why we can afford these operations will be become clearer in chapter 6, the main idea is that all the operations we can afford will take an insignificant amount of time compared to updating the suffix array.

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 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.

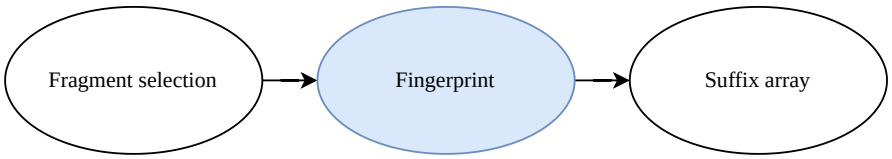
When a file is opened, the LSP client sends a `textDocument/didOpen` message to the server, which finds the relevant document D in the index, and sets the following fields:

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

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 $D_1.start > D.start$ 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 Computing edit operations

Now that the fingerprint has been updated, we could build the suffix array from scratch and already see a substantial 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 to further improve efficiency.

The input to the dynamic suffix array algorithm is a set of edit operations. An edit operation can be either deleting, inserting or substituting a set of consecutive characters in the fingerprint. The dynamic suffix array algorithm will take each edit operation, and update its suffix array to reflect the new state of the fingerprint. Therefore, the first problem to solve is to determine what exactly has changed in the fingerprint.

There are a couple of approaches we could take to determine the edit operations. The simplest is that whenever a file has changed, regard the entire file as changed, and perform a delete operation which removes the entire fingerprint of that file, and then an insertion, which inserts the new fingerprint of that file. This is a very simple approach, but leads to an unnecessary size for the edit operations. The fingerprint of the file is likely to be very similar to the previous version, therefore deleting and then inserting the entire files fingerprint could be a lot more work than

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

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

$$M[i][j] = \begin{cases} M[i-1][j-1] & \text{if } S_1[i] = S_2[j] \\ 1 + \texttt{Min}(M[i-1][j-1], M[i][j-1], M[i-1][j-1]) \end{cases}$$

Figure 5.1: Edit distance recurrence

it needs to be. However, this idea that we can always reduce the number of operations to only two, a deletion and an insertion, will be useful in our final algorithm.

Another approach 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 according to this range. However, this approach tightly couples the algorithm to LSP and the scenario where we know the exact ranges of each change. Also, we might do unnecessary amounts of operations if we do multiple edits, since some operations could cancel each other out, for example by inserting and then deleting the same text.

A better approach is to determine the changes of the fingerprint using an edit distance algorithm. An edit distance algorithm is an algorithm which calculates the distance between two strings S_1 and S_2. Distance between two strings is 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 [40]. 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. The recurrence in equation 5.1 shows how the edit distance matrix is filled and 5.1 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 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

		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 5.1: Edit distance matrix for REPUBLICAN → DEMOCRAT

equal, the operation can be ignored. For example in table 5.1, the first operation is to substitute `R` with `D` at position 0. Afterwards `P` and `U` is deleted at position 2. Then the `B` which is now at position 2 is substituted by `M`. This continues with more substitutions until we finally have `DEMOCRAT`.

Aggregating edit operations

In the next phase we will feed the edit operations into an algorithm which dynamically updates our suffix array based on those operations. However, this algorithm will be more efficient if the operations are combined to singular inserts, deletes or substitutes of strings more than one character. The edit distance algorithm outputs only single character operations, meaning we insert, delete or substitute a single character at a time. We therefore want a way to combine these operations to "larger" operations.

We define an EditOperation with the following record:

```
record EditOperation {
    OperationType type,
    char[] chars,
    int position
}
```

where type is either an insert, delete or substitute.

One way to combine operations is to find operations of the same type which are sequenced in the matrix. For example in table 5.1, we have two consecutive delete operations, where `P` and `U` is deleted at position 2. These two operations could be combined to a single delete operation of two characters.

The idea for the algorithm which computes the edit operations is to traverse the optimal matrix path backwards and for each operation we either append to the current operation if possible, or start a new operation. We can continue the current operation if the next operation is of the same type, and the next operation has the same position as the current operation. For example in table 5.1, we have two delete operations in a sequence at position 1 where `P` and `U` is deleted. The first operation we encounter is the deletion of `U`. At this point we create a new `EditOperation` with

position 1 and `U` in the `chars` array. The next operation is also a delete operation at position 1, so we add the `P` to the list of characters for the operation to delete. The next operation is a substitute at position 0, so we cannot continue the delete operation, and a new substitute operation is created instead. Similarly, at position 2 to 5, we substitute `BLIC` with `MOCR`. This operation is computed in a backwards fashion similarly to how we did the deletion, except that the position of the operation is decremented for each character we add to it.

Note that this algorithm is not an optimal algorithm, as this is a trade-off between processing many characters in few operations, versus processing many operations with few characters. As mentioned earlier, one could always reduce the solution to be only two operations, deleting the whole string, and inserting the new string. This is only two operations, but likely processes more characters in total compared to our algorithm. This trade-off will become apparent when discussing the dynamic suffix array updates.

Optimizing memory usage and operations

A problem with the above solution is the memory usage of the matrix. It is not feasible to input the entire old and new fingerprint into an edit distance algorithm, as the full fingerprint can have millions of symbols, and the old and new fingerprint is likely approximately the same size. This would require a matrix which is too large to fit in memory. We will use a few techniques to reduce the memory usage of this algorithm without compromising on the time complexity.

The first technique is to not input the whole fingerprint. We can drastically reduce the size of the input by only comparing the old and new fingerprint of the document D which has been edited. D stores its previous and current fingerprint, and whenever it is edited, we can compute the edit operations of D, with its previous and current fingerprint as input. However, the position of each edit operation of D will not have the correct position relative to the entire fingerprint, so $D.start$ is added to the position of each edit operation to correct this.

Another optimization we can do to reduce the size of the matrix is to remove the "trivial" part at each end of our matrix. If we compare two strings which have many similar characters at the beginning or end of our string, we know that these will not equate to any edit operations in the matrix, as they will simply be diagonal moves (substitutes) where the characters are already equal. In table 5.2 we see the edit matrix for the input `FASCINATING` and `FINISHING`. The two words share the common prefix `F` and the common suffix `ING`. If we examine the edit distance values, we see that the highlighted green matrix starts at 0 in the top left, and ends with 6 in the top right, which is the same final values as in the full matrix. Also, we see that there are no edit operations being added outside the green matrix. Using this knowledge, we can see that we only need to compare the strings `ASCINAT` and `INISH`, which would give us the exact same edit distance. We can also get the exact same edit operations, as long as we account for the starting offset of the original string, so that the operations have the correct positions. Algorithm 11 shows how two input strings can be minimized for usage in the edit distance algorithm. After getting the edit operations from the algorithm, the `startOffset` is added to the position of each operation to account for the offset of the minimized matrix.

The two optimizations drastically reduce the memory and time usage of the edit distance algorithm, but in cases where a very large file is edited, and the beginning and end of the old and new fingerprint do not match, we can still encounter instances of the matrix being too large to fit in memory. The problem is that the matrix size has a polynomial growth in terms of the fingerprint size. This is because the old fingerprint is of size n and the new fingerprint is of size

		F	I	N	I	S	H	I	N	G
	0	1	2	3	4	5	6	7	8	9
F	1	0	1	2	3	4	5	6	7	8
A	2	1	1	2	3	4	5	6	7	8
S	3	2	2	2	3	3	4	5	6	7
C	4	3	3	3	3	4	4	5	6	7
I	5	4	3	4	3	4	5	4	5	6
N	6	5	4	3	4	4	5	5	4	5
A	7	6	5	4	4	5	5	6	5	5
T	8	7	6	5	5	5	6	6	6	6
I	9	8	7	6	5	6	6	6	7	7
N	10	9	8	7	6	6	7	7	6	7
G	11	10	9	8	7	7	7	8	7	6

Table 5.2: Edit distance matrix for FASCINATING → FINISHING. Blue is optimal path, green is minimized matrix

Algorithm `EditDistanceMinimizeStrings`(S_1, S_2)

for i **from** 0 **to** `Min(Len(`S_1`),Len(`S_1`))` **do**
  if S_1[i] $\neq$ S_2[i] **then** **break**
end
startOffset $\leftarrow$ i

s1End $\leftarrow$ `Len(`S_1`)` $-$ 1
s2End $\leftarrow$ `Len(`S_2`)` $-$ 1
while s1End $\geq$ startOffset **and** s2End $\geq$ startOffset **and** S_1[s1End] $=$ S_2[s2End] **do**
  s1End $\leftarrow$ s1End $-$ 1
  s2End $\leftarrow$ s2End $-$ 1
end

miniS1 $\leftarrow$ S_1[startOffset...s1End]
miniS2 $\leftarrow$ S_2[startOffset...s2End]

return (miniS1, miniS2, startOffset)

Algorithm 11: Minimize strings for edit distance algorithm

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

Table 5.3: Edit distance matrix with minimal memory usage

m, where $n \approx m$, which requires an $n \times m$ size matrix to calculate the edit operations. For example if a Java file contains 3000 lines, the number of tokens can exceed 10000, which would require approximately a 10000×10000 size matrix, which is approaching a memory usage which too large.

A solution to this problem is to reduce the required memory from the polinomial $O(n \times m)$ memory usage, to a linear $O(n)$ memory usage. A stepping stone towards such a solution is an observation attributed to Ukkonen, which reduces the required memory to linear growth in the size of either of the strings [38]. The observation shows that in order to compute the next row/column of the edit distance matrix, we only need the previous row/column. For example in table 5.3, the fifth row has been computed using only the fourth row. This is done by first computing the left-most index of the fifth row, which is always one more than the previous row. Afterwards, we can compute the other elements of the row from left to right, with the same recurrence, shown in equation 5.1. This is possible because we always know the above, left, and top-left elements of the current index. This can be implemented as two arrays, which holds the previous and current row.

This change allows us to compute the edit distance in linear space, but now the problem is how to find the actual edit operation. This is not possible, because we don't have the entire matrix available to traverse anymore. However, this can be solved using Hirschberg's algorithm [16]. Hirschberg's algorithm is an algorithm which can compute the edit operations of two strings in the same time complexity as the Wagner-Fischer algorithm, but uses only linear space in the size of either of the input strings.

The first insight we need for this algorithm is that there is at minimum one edit operation on each row/column of the edit distance matrix. This is intuitive, because in order to "travel" from the top-left to the bottom-right of the matrix, the path needs to visit at least one cell from the top to the bottom, visiting all the rows, and from the left to the right, visiting all the columns. Hirschberg's algorithm uses this insight to compute one position of the optimal path at a time, which it finds in the middle row between two already known positions of the path. Table 5.4 shows an example of how the positions are found.

Initially, we know that the top-left index, and the bottom-right index of the matrix are guaranteed to be part of the optimal path, since those positions are the starting and ending point of the

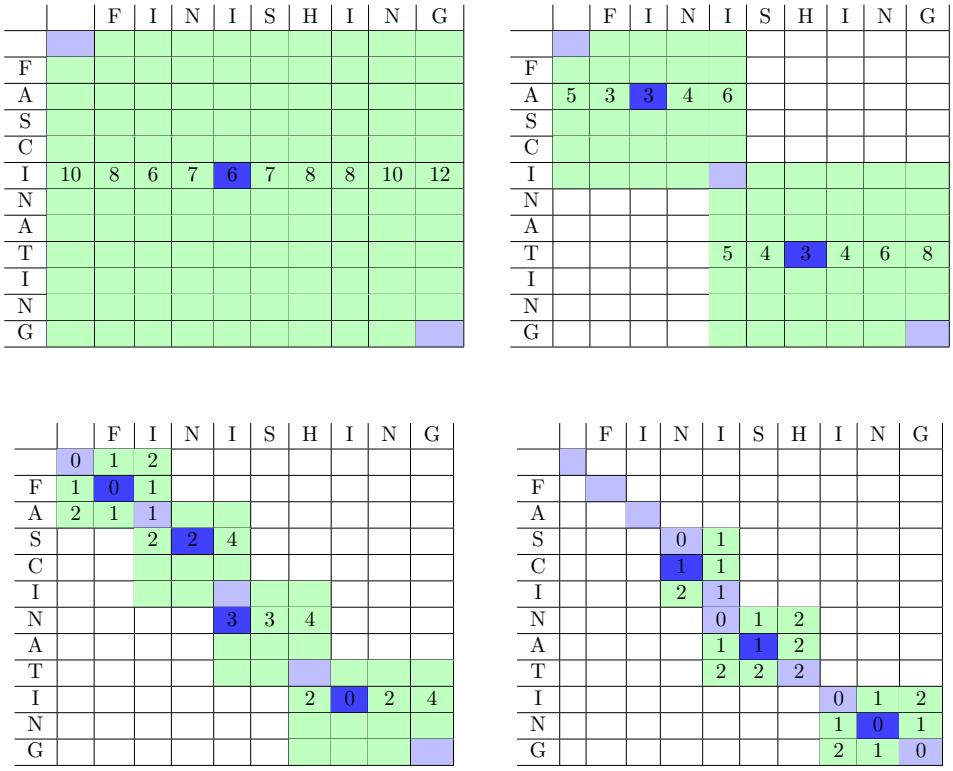


Table 5.4: Hirschberg's algorithm. Blue cells are part of the optimal path, dark-blue cells are new positions.

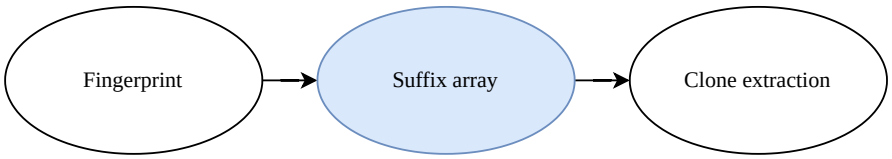
path. The next position to find is in the middle row of the matrix. Determining which position in the middle row should be selected is done by performing the edit distance recurrence twice, once from the beginning to the middle row in the same fashion as the original algorithm, and once in reverse, from the bottom-right to the middle row. Both of the recurrences can run in linear space, because we only need to store two rows at a time. Summing the two results for the middle row gives us an array which can intuitively be understood as how long the minimal path which goes through the corresponding position in the row is. Since we know that there is at least one of the positions in this row which has to be part of the optimal path, the minimum value in this array corresponds to one position which has to be part of the optimal path. Once we know the middle position which is part of the optimal path, we can recursively call the function twice, once with the top-left and middle as input, and once with the middle and bottom-right as input. The base-case of the recursion is when the size of the matrix is linear in the size of either of the strings, in which case we call the Wagner-Fischer edit distance recurrence with a linear space complexity. Note that there can be multiple minimum values in the row where a position is selected, which corresponds to the fact that there can be multiple optimal paths through the matrix.

Each position which is a part of the optimal path is stored as they are found, and can then be traversed backwards to determine the edit operations similarly to traversing the matrix. For example if we iterate over the list of positions backwards, we will know that if the first position is at x, y and the second position is at $x - 1, y$, that corresponds to a delete operation, just as in the matrix-based algorithm.

Using Hirschberg's algorithm, we are now able to compute the edit operations for very large files without memory usage issues.

A final problem we should be aware of is that in some instances, the number of edit operations can be very large. As mentioned, our algorithm which aggregates edit operations together is not optimal, and could likely be improved to reduce the number of edit operations when we allow more than one character in each operation. Therefore, in instances where the edit distance algorithm returns so many edit operations that it is likely very slow to use them in the next phase, we will instead reduce the solution to only two operations, deletion and insertion of the entire minimized string. In practice this optimization can often avoid large spikes in the running time of the next phase.

5.5 Dynamic suffix array updates



Now that we know how the fingerprint has changed after an edit, the next phase is to input the edit operations into an algorithm which dynamically updates the suffix array. The algorithm discussed in this section will be run once for each edit operation.

Constructing/updating the suffix array is the most time-consuming phase of the algorithm, so

Order	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**

(a) S = BANANA$

Order	Cyclic-shift
7	$BABNAN**A**
6	A$BABNA**N**
1	ABNANA$**B**
4	ANA$BAB**N**
0	BABNANA**$**
2	BNANA$B**A**
5	NA$BABN**A**
3	NANA$BA**B**

(b) S = BABNANA$

Table 5.5: BWT for string before and after insert

the goal for the dynamic update is to take the previous suffix array and an edit operation, and compute the new suffix array faster than it would take to build the suffix array from scratch. We will also in the following section convert the suffix array into a dynamic structure, which allows insertions, deletions and increments without requiring linear time.

The algorithm used is the algorithm by Salson et al. [34]. This algorithm is based on the 4-stage algorithm for updating a Burrows-Wheeler transform (BWT), also by Salson et al. [33]. Recall that the BWT is a transform on an input string which is often used for compression and text-search purposes. The BWT is also reversible by using the LF function to compute the original string in reverse. The LF function is computed for a position i such that $LF(i) = rank_{BWT[i]}(i) + C[BWT[i]]$ where the array C contains for any character c, how many characters are lexicographically smaller than c. This function can intuitively be used in the context of the BWT to allow us to move from CS_i to CS_{i-1}. Also recall that the suffix array and the BWT of a string is tightly related to the point where one can compute the BWT from the suffix array. Any updates to the BWT will therefore lead to similar updates to the suffix array, so the algorithm which updates the BWT can be used to determine how the suffix array changes as well.

Updating BWT on insertion

First we will observe which parts of a BWT changes when we insert a single character. Note that we will only consider changes to the BWT, meaning the final character in each shift, not the whole cyclic-shift. We will consider an insertion of a character c, inserted at position i. We know that we will have exactly one more CS for the string, which starts with the inserted character c, at position i. This CS ends with the character at position $i-1$. We also know that exactly one CS will change its last character, which is the cyclic-shift previously starting at position i, but is now at position $i+1$ after the insertion. This new cyclic-shift will now end with the new inserted character, c. These are the only changes that happen to the characters in the BWT, but the ordering might not be correct anymore.

For the string `BANANA$`, let $c =$ `B`, and $i = 2$, which results in the string `BABNANA$`. In table 5.5 we can see that some substrings of the BWT is preserved, such as `$AA` and `AN`, but some other changes have occured. In table 5.6 We see that the new CS_2 is `BNANA$BA`, and the changed CS_3 is `NANA$BAB`. We can find where these cyclic-shifts are located by first looking up i in the ISA, ISA[2] = 6. This is the location of CS_3 (after the insertion), where the final character is updated to the inserted character `B`. To find the position where the new cyclic-shift should be inserted, we

Order	F	L
6	$	A
5	A	N
3	A	N
1	A	B
0	B	$
4	N	A
2	N	A

(a) Original BWT

Order	F	L	
7	$	A	
6	A	N	
4	A	N	
1	A	B	
0	B	$	
2	**B**	**A**	Inserted
5	B	A	
3	**N**	**B**	$A \rightarrow B$

(b) After change and insert

Order	F	L	
7	$	A	
6	A	N	
1	**A**	**B**	↕
4	**A**	**N**	
0	B	$	
2	B	A	
5	B	A	
3	N	B	

(c) After reordering

Table 5.6: BWT for string before and after insert

can use the LF function to move from the CS_3 to CS_2. Computing $LF(6)$ after the substitute gives us 5, which is where the new CS_2 should be inserted in the BWT. When we insert in the BWT, we only need to consider the final character of CS_2, which in this case is `A`, the character which was previously substituted.

The final stage of the algorithm now that all the elements are present, is to rearrange the cyclic-shifts which have changed their lexicographical ordering. It is possible that the cyclic-shifts change their ordering because of the new character. For example `ANANA$B` < `ANA$BAN`, but `ABNANA$B` > `ANA$BABN`. A useful observation is that only CS_j where $j \leq i$ will have their lexicographical ordering changed. This is because only in these cyclic-shifts the inserted `B` comes before the `$`. Since the `$` is the smallest lexicographical character, and no two cyclic-shifts will have `$` in the same location, we know that the lexicographical comparison of two cyclic-shifts will never go past the `$`. Therefore, only the cyclic-shifts where the inserted character comes before the `$`, have the possibility of being moved.

We know that only CS_j where $j \leq i$ can have their ordering changed, and we have already inserted CS_i at the correct position (the new cyclic-shift). We will store two positions, *pos* and *expected*, where *pos* is initially the position of CS_{i-1}, which was found before any changes were made to the BWT. *pos* is therefore the actual position of CS_{i-1}, *expected* is the expected position of CS_{i-1}, which is computed by computing $LF(insertionPoint)$ where *insertionPoint* is the position where the new CS_i was inserted. With the knowledge of where the cyclic-shift of order $i-1$ is, and where it should be, we can move the cyclic-shift to that position. In the example, we have that the $expected = 3$, and $pos = LF(5) = 2$ after the insertion gives us the expected location of the cyclic-shift of order 1. Therefore, we remove the cyclic-shift at position 3, and insert it again at position 2.

Before performing the row-move, we also compute $newPos = LF(pos)$ which will give us the position of the cyclic-shift of order $i-2$. After the row-move, we can continue comparing the position and expected position by updating $pos = newPos$, and $expected = LF(expected)$. In the example this would update both *pos* and *expected* to 4. Since the expected position and actual position is the same, the algorithm is done, and we know that every position of the BWT is now in the correct location. We know that there will be no more row-moves for any other CS down to order 0 because $LF(pos) = LF(expected)$, which would be true for all future iterations as soon as $pos = expected$

```
Algorithm UpdateSuffixArrayInsert(SA, ISA, BWT, i, ch)
    posFirstModified ← ISA[i]
    previousCS ← LF(BWT, i)

    storedLetter ← BWT[posFirstModified]
    BWT[posFirstModified] ← ch

    insertionPoint ← LF(BWT, i)
    if storedLetter < ch then insertionPoint ← insertionPoint + 1

    // Insert storedLetter in BWT at pointOfInsertion
    Insert(BWT, insertionPoint, storedLetter)

    // Insert i in SA at pointOfInsertion, increment all values ≥ position
    Insert(SA, insertionPoint, pos)
    IncrementGreaterThan(SA, pos)

    if insertionPoint ≤ previousCS then previousCS ← previousCS + 1

    pos ← previousCS
    expected ← LF(insertionPoint)
    while pos ≠ expected do
        newPos ← LF(pos)

        // Delete value at pos and reinsert at expected in BWT and SA
        MoveRow(pos, expected)

        pos ← newPos
        expected ← LF(expected)
    end
```

Algorithm 12: Update BWT and suffix array when inserting a single character

Updating SA on insertion

The suffix array is updated similarly to the BWT. When we insert the new cyclic-shift at position 5 in the BWT, we similarly insert the value 2 at position 5 in SA. We insert the value 2 because the new suffix which corresponds to the new cyclic-shift starts at the position where the new character was inserted, which was 2. When the new cyclic-shift is inserted, we are now in an invalid state for the suffix array, because we have two elements with the value 2. Note that a suffix array is always a permutation of the values $(0..n)$ where n is the number of characters in the input. Since we have inserted a new suffix in the suffix array which is the 2nd smallest suffix, all the previous suffixes which had a value $j \geq 2$ is now incremented. After incrementing all these suffixes, we are now in a valid suffix array state, but similarly to our BWT, some suffixes may have had its ordering changed. For every move-row operation in the BWT, the elements in SA are reordered in the same fashion.

Algorithm 12 dynamically updates a suffix array and BWT when a single character is inserted.

This algorithm can easily be extended to insert a string instead of a single character by adding a loop which continuously updates the `pointOfInsertion` to the previous CS with the LF function, and inserts all the characters at that position into the BWT and SA backwards. This is more efficient than calling the single character algorithm multiple times, because the reordering stage is only performed once. The details for insertions/deletions of multiple characters is covered in the original paper [34].

Deletion

A deletion of a single character is a similar procedure as an insertion, as it is simply reversing the substitution and insertion, and then performing the reordering stage again. If we have the string `BABNANA$`, and delete the `B` at position 2, We know that there will be exactly one CS removed. This is CS_2 which starts with the deleted character `B`, and ends with the character before it, `A`. There is also a single CS which will have its final character changed, CS_3, because the final character `B` will be deleted. The final step is to again perform the reordering stage for CS_j where $j \leq 2$, as these are the only cyclic-shifts which can have their lexicographical ordering changed.

The algorithm which performs the deletion at position i will do the deletion and substitution in the opposite order. First we find both the CS which will have its character substituted in the BWT, and the CS which will have its character deleted in the BWT. The character to be substituted will be found with $substituted = ISA[i + 1]$, and the character to delete will be found with $deleted = LF(substituted)$. $BWT[deleted]$ is then deleted, and $BWT[substituted]$ is substituted with the character that was deleted. Finally, the reordering phase is performed in the same fashion, where pos is set to the position of the original CS_{i-1} and $expected$ is set to $LF(substituted)$

In our example $substituted = ISA[3] = 7$, $deleted = LF(7) = 5$ and $pos = LF(5) = 2$. Then $BWT[deleted]$ is deleted, and $BWT[substituted]$ is set to the deleted character. Then $expected = LF(6) = 3$. Note that $substituted$ was decremented after the deletion because the deletion moved that CS one position up. When the reordering stage happens, we have $pos = 2$ and $expected = 3$, so we perform the reordering in the BWT and SA in the same fashion as previously, and the algorithm will again terminate after one iteration of the reordering phase.

Substitution

Substitution operations are also possible and covered in the original paper, but due to time constraints, these were implemented as a deletion followed by an insertion, which is less efficient than a single operation, since the reordering stage is done twice.

LF function

The LF function is called multiple times for any operation which is done to update the suffix array. Therefore, it is important to be able to efficient compute LF at any point in time. The naive approach of linearly iterating through the BWT to compute the $rank$ and number of smaller characters is too slow. Recall that $LF(i) = rank_{BWT[i]}(i) + C[BWT[i]]$ where $rank_{BWT[i]}(i)$ is the rank for the character $BWT[i]$ at position i in the BWT, and $C[BWT[i]]$ counts the number of lexicographically smaller characters than $BWT[i]$ in the BWT. We therefore need a data structure for $rank$ queries on the BWT, and a data structure which stores the number of smaller characters of each character in the BWT.

For the *rank* queries, recall that the wavelet matrix was a data structure which could efficiently compute *rank*/*select* queries for strings. We will use the wavelet matrix to store the BWT and perform efficient *access* and *rank* queries on it as needed. With the wavelet matrix, we do not need to store the BWT itself, as we can simply query the wavelet matrix to access characters of the BWT. We also do not need to store the full fingerprint, since we can access characters in the wavelet matrix and if we need to access the characters in a sorted order, we can use the LF function. We decided to use the wavelet matrix instead of a wavelet tree as it has been shown to have faster *rank* queries, and is more memory efficient for larger alphabets. Our alphabet can be quite large compared the standard applications of *rank*/*select* data structures (such as DNA analysis). The wavelet matrix is also generally simpler to implement, and can be dynamically updated in a simple manner. Each bitset in the wavelet matrix is a dynamic bitset where we can efficiently insert and delete characters as needed when the size of the BWT grows/shrinks.

When inserting a character c into the wavelet matrix at position i, we insert the first bit of c in level 0 in the matrix at position i. The next bit of c is then inserted in level 1 in the matrix at position $rank_x(WM[0], i)$ where x is the value of the previous bit.

A complication with the wavelet data structures is to update the data structure as the alphabet increases in size. In our case the alphabet size will at some point increase to a size where elements need an additional bit. For the wavelet matrix, this can be solved by simply inserting a new row at the beginning of the matrix (level 0), where the bitset consists of all zeroes. Afterwards, the new element is inserted in the normal fashion, which should be the only element with a 1 bit in the first position.

TODO: Algorithm to insert/delete in wavelet matrix

The C array can be implemented in two ways. Either $C[c]$ holds the number of smaller character than c in the input, or $C[c]$ holds the number of occurrences of c. There are trade-offs with either implementation, as storing the number of smaller characters requires a linear scan whenever a character is inserted/deleted, while storing the number of occurrences requires computing the number of characters whenever the LF function is called. We have chosen the approach of storing the number of occurrences, as updating the array is simpler in cases where the alphabet size increases. When we later update LCP values, we will see that there is a possibility that choosing the option of storing the number of smaller characters in C is faster.

5.6 Dynamic extended suffix arrays

A major bottleneck when inserting/deleting elements in the suffix array is that when an element is removed, all elements after it needs to be moved by one index so that the gap is closed. This is also the case in the reordering stage, where all the elements between the old position and new position of an element needs to be moved by one position. In addition, since a suffix array is always a permutation from 0 to n where n is the number of elements in the input, we need increment/decrement all elements greater than or equal to the element which was inserted/deleted. For example, if we insert a new element into the suffix array which indicates that a new suffix is the ith smallest, then naturally the previous suffix with value i in SA should now have the value $i + 1$, which applies to all other suffixes lexicographically greater than the new suffix as well. In a standard representation for a suffix array (an array), this would require either a linear scan through the entire SA, or a linear scan through ISA from position i, which is also linear in the worst case. Multiple linear scans through the SA would make the algorithm slower than the

linear time SACA when the amount of inserted/deleted character increases.

In this section we will introduce a data structure called the dynamic extended suffix array. This data structure consists of a dynamic permutation, which facilitates insertion/deletion of elements in a permutation from 0 to n, without linear scans to close gaps or incrementing/decrementing elements. The data structure stores SA, ISA and LCP and is inspired by the data structure introduced for the same purpose by Salson et al. [34]. The data structure used by Salson et al. uses the same dynamic permutation tree, but reduces the amount of values which is stored to reduce memory consumption, but also slows down the access time of elements in the permutation. Our data structure will not be compressed, which allows faster access to arbitrary elements, and will be extended to also include the LCP values as well. Therefore, this data structure encompasses the entire extended suffix array.

The data structure consists of two balanced binary trees, with pointers between elements in one tree to the other. Figure 5.2 shows the two trees which are labeled the A tree and the B tree. Note that the trees do not actually hold the values displayed in the figure, the values in each node of the figure only represents the inorder position of each node, which is called the inorder rank of the node.

To construct a permutation such as $[6, 5, 3, 1, 0, 4, 2]$, for each element, insert a node in each tree. After the trees have all the nodes in them, pointers are added from a node in A to a node in B and vice versa (doubly linked). If an element in SA has a value i at position j, pointers will be added between the node in A with inorder rank i, and the node in B with inorder rank j.

Now the tree is complete, but we can also insert new nodes into the data structure as the suffix array grows. When a new value j is added at position i, a node is inserted into A such that it has the inorder rank of i, and a node is inserted into B such that it has the inorder rank of j. For example if we wanted to insert a new value 4 at position 1 in the permutation in figure 5.2, we would insert a new node in A so that it is the right child of the node labeled 0, and then insert a new node in B which is the left child of the node labeled 4. The trees can intuitively be understood such that the inorder values in A represent the indices of the permutation, and the inorder values in B represent the values in the permutation. Therefore, if we look at a node in A with inorder rank i, and follow the pointer to a node in B with inorder rank j, this means that the permutation has the value j at position i.

Deleting a node at position i is similar, but we will instead find the node with a rank of i in A, then delete that node and the node it points to in B. As the trees are implemented as balanced trees, traversing, inserting and deleting nodes takes $O(\log n)$

An essential property of this tree is that when a new value j is inserted into the permutation at position i, all indices $\geq i$ and values $\geq j$ is incremented by 1. This is because all the nodes which had an inorder rank greater than or equal to the new nodes in each respective tree has now been incremented by 1 simply because the structure of the tree has changed. No additional work is required to increment the elements.

The part we are still missing is how to find a node with inorder value i, and how to determine the inorder value of a given node. We cannot afford to traverse all the nodes in an inorder fashion to find a node with a specific inorder rank. Therefore, the trees are implemented as order-statistic trees [10, p. 340]. An order-statistic tree is a tree where each node x stores an additional integer $size$, which contains the number of nodes in the subtree rooted at x. This allows us to easily

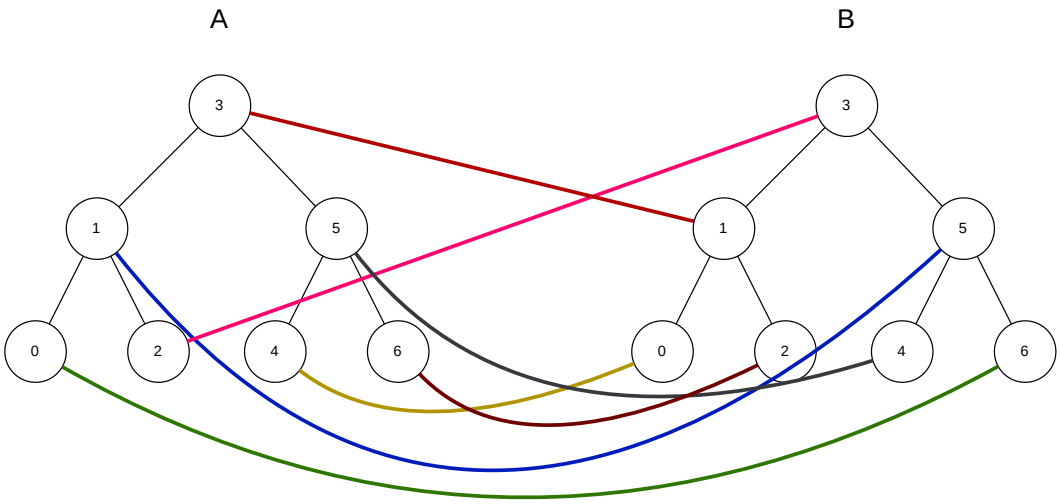


Figure 5.2: Dynamic permutation for the permutation $[6, 5, 3, 1, 0, 4, 2]$. Colors have no meaning except to make lines clearer

determine how many nodes is in the left subtree of a given node, and use this to determine where the node with a certain inorder rank is located. If we wanted to find the node in the A tree of figure 5.2 with inorder rank 4, we would start at the root, see that the left subtree of the root contains 3 nodes. Therefore, the roots inorder rank is 3. With this information we know that we need to traverse to the right child, and look for the node with inorder rank $4 - 4 = 0$ in that subtree (since we have already seen 4 nodes). Since the node labelled 5 has a rank of 1 (in its own subtree), we know to traverse left, and at that point we have reached the correct node.

Algorithm 13 shows how to find a node in a tree by its inorder rank, and algorithm 14 shows how to determine the rank of a given node. The algorithm to find the value at a certain position is simple with these algorithms. To find the value in the permutation at position i, find the node with inorder rank i in A, follow its pointer to a node in B, and then find the inorder rank of that node, which is the value at position i in the permutation. We can also find the position of any value j in the permutation by performing this algorithm in reverse, by instead starting at the B tree, finding a node with rank j, then following its pointer to a node in A and returning that nodes inorder rank.

With the ability to both find a value at a given position in the permutation, and also finding the position of a given value, we can use this data structure to represent the suffix array and inverse suffix array, instead of a simple array. Inserting, deleting and accessing a node in this data structure takes $O(\log n)$ time, and we no longer need a linear scan through the data structure to increment/decrement values, as values are automatically incremented/decremented when a new element is inserted/deleted.

5.7 Dynamic LCP array updates

Now that we have a dynamic structure to represent SA and ISA, we can extend the data structure to also contain the LCP array, and efficiently update the LCP values as well.

```
Algorithm GetNodeByRank(root, rank)
    current ← root
    while current ≠ null do
        currentRank ← current.left.size
        if currentRank = rank then
            break
        else if currentRank > rank then
            current ← current.left
        else
            current ← current.right
            rank ← rank − currentRank + 1
        end
    end
    return current
```

Algorithm 13: Find the node in a tree with a given inorder rank

```
Algorithm GetRankOfNode(node)
    rank ← node.left.size
    while node.parent ≠ null do
        if node.parent.right = node then
            rank ← rank + node.parent.right + 1
        end
        node ← node.parent
    end
    return rank
```

Algorithm 14: Find the inorder rank of a given node

Since every node in the A tree of our data structure correlates to an index, we can easily extend the data structure to contain LCP values by setting a value in each node of the A tree. This would make the A tree work as a balanced binary tree for the LCP values, and if an SA value at position i is deleted, the LCP value stored in the respective node is also deleted.

In the initial construction of the dynamic permutation, LCP values would be inserted together with the nodes. When dynamically updating the LCP values as suffixes are inserted/removed, the procedure is more complicated. Salson et al.[34] also describe a procedure for updating the LCP values after an insert/delete operation, which we will use.

When inserting a character into the input string, the suffix array changes by inserting a new value, and afterwards moving values from one position to another (reordering stage). Similarly, a deletion in the suffix array consists of deleting a value, and performing the same reordering operations. The moving of a value from position i to j can be decomposed into a deletion at position i followed by an insertion at position j. When updating the LCP array we therefore need to consider how insertion of a value and deletion of a value in SA affects the LCP values.

The idea for this algorithm is to keep track of all positions which could possibly have its LCP value changed, and update the LCP values after the SA is fully updated. There are four different cases where an LCP value at a position i could be changed:

Algorithm `DynamicPermutationAccess(`*permutation, i*`)`

```
aNode = GetNodeByRank(permutation.A.root, i)
bNode = aNode.pointer
return GetRankOfNode(bNode)
```

Algorithm 15: Get value at position i in a permutation

1. A suffix was inserted at position $i-1$ in SA, which is the new suffix that the suffix at position i should compute its LCP value for.

2. The suffix at position $i-1$ was deleted in SA, making the suffix at position $i-2$ the new suffix that the suffix at position i should compute its LCP value for.

3. A character was inserted/deleted in the middle of the LCP for the suffix at position i in SA, which shortens the LCP value between it and the suffix at position $i-1$, and the LCP value between the suffix at position i and $i+1$.

4. A character was inserted/deleted at the end of the LCP for the suffix at position i in SA, which could potentially extend the LCP value between it and the suffix at position $i-1$, and the LCP value between the suffix at position i and $i+1$.

To cover the first two cases, we will store a dynamic bitset where a set bit at position i indicates that at some point, the LCP value at position i needs to be updated.

For an insertion at position $i-1$ in SA (case 1), we will set the bit at position i, as the suffix it compares its LCP to has changed. We will also insert a set bit at position $i-1$, as the newly inserted suffix also needs to compute its LCP value with the suffix at position $i-2$.

Similarly, for a deletion at position $i-1$ in SA (case 2), we will set the bit at position i, as the suffix it used to compare its LCP to has been deleted. We will also delete the bit at position $i-1$, as the suffix at that position has been deleted.

After setting these bits while performing an insert/delete operation in the SA, the bitset now contains all the positions of LCP values which need to be updated for the first two cases. The other two cases are more complex to find the positions of. Instead of inserting the positions of these suffix into the bitset, we will iterate over the suffixes which could possibly have their LCP value changed. Iteration over every suffix and recomputing the LCP value takes linear time in the size of the fingerprint and is therefore too slow for this algorithm. However, we can reduce the number of suffixes we need to update by using a series of observations.

The first observation is that we only need to consider suffixes where the insertion/deletion of a character at position i has changed the suffix. Those are the suffixes which start at a position j where $j \leq i$. Other suffixes cannot change, as the insertion/deletion happened before the starting position of the suffix. For a suffix at position j which has changed, two LCP values can change, the values at position k and $k+1$ where $SA[k] = j$. These two LCP values can change, because at position k, the suffix at position $SA[k]$ is compared with $SA[k-1]$, and at position $k+1$, the suffix at position $SA[k+1]$ is compared with $SA[k]$. Since both of these LCP values depend on the suffix at position $SA[k] = j$, they can potentially change when the suffix is changed. The positions in SA which possibly can be changed can be found using the LF function. In the final

phase of the suffix array update algorithm when we have inserted/deleted an element at position i, we are reordering suffixes at position $j \leq i$. These are the suffixes we are looking for, but we can skip all suffixes which are reordered, as they are already marked to be computed in the bitset for the previous cases. Therefore, the variable *pos* points to the next suffix we need to consider at the end of algorithm 12.

Another useful observation when iterating over suffixes is that after an insertion/deletion, no two suffixes will have the insertion/deletion occur at the same position. This is true because no two suffixes start on the same position in the input, and an insertion/deletion at a position i in the input can therefore never correspond to an insert/deletion at the same position for two different suffixes. With that in mind, another useful observation is that given two suffixes with an LCP value of p, any change inside the range of the LCP in either of the suffixes, will change p.

The idea is to determine if the insertion/deletion is out of range to possibly affect the LCP value of a suffix. Recall that the LCP value for a suffix at position $i-1$ is at minimum one less than the LCP value for the previous suffix at position i, as discussed in chapter 4. We can determine that it is impossible for the LCP value at position $i-1$ to change, if neither of the LCP values which is affected by the suffix at position i changes, unless the insertion/deletion happens at position $i-1$. This is explained by the fact that at minimum, the LCP of the suffix at position i covers every character of the LCP of the suffix at position $i-1$ except for the first character. If the first character was the position of the insertion/deletion, it can change the LCP value, but that case is already handled by the positions stored in the bitset. Since the suffix at position $i-1$ did not update, this holds recursively for the next suffix at position $i-2$ as well. Therefore, as soon as we find a single suffix at position i which does not lead to an LCP value update, this will recursively hold for all suffixes at position $j < i$.

With this in mind, the algorithm to update the LCP array has two stages. Given the bitset of positions which need to be updated in correlation with the first two cases, we perform a *select* operation to continuously find the location of set bits in the bitset, and use their position to determine which positions in the LCP array need to be computed. After all the positions in the bitset has been updated in the LCP array, we move on the last two cases. From the position of *pos*, which was computed during the suffix array update, we continuously call the LF function on *pos* to find the previous suffix. At each position, we update the LCP value of the suffix at position *pos* and $pos + 1$, as these two LCP values are both affected by the suffix at position $SA[pos]$. When we encounter the situation where both LCP values at position *pos* and $pos + 1$ did not lead to a change in LCP values, the algorithm is finished and the LCP array is updated. Algorithm **??** shows how the LCP array is updated, given the bitset of updated positions which is built during algorithm 12.

The actual computation of LCP values is also more complex than it seems. In the dynamic detection algorithm we are not storing the entire fingerprint, we only have the BWT stored in the wavelet matrix, and the LF function allows us to move from one character to the previous in the original input text. When we compute LCP values, we need to be able to compare characters in two suffixes from start to potentially end of the input, therefore we need to be able to move from one character to the next, which is the inverse of the LF function.

The inverse LF function for a position i computes the next cyclic-shift by first determining what character is in the first column of the CS at position i in the sorted cyclic-shifts, and its *rank* in the first column. With the character and rank of that character, we can perform a *select* on the BWT for that character and rank, which will give us the location of where that character

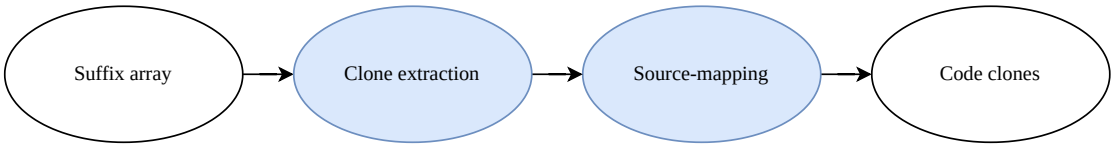
is located in the last column (BWT) which is the next CS. This is the next cyclic-shift because it is the cyclic-shift where the character which was previously in the first column, but is now cycled one character to the left, which corresponds to the next CS. To find which character is in the first column at position i, recall that the F column in the sorted cyclic-shifts just contain all the characters of the input in sorted order. We can therefore use the C to determine which character is located at exactly position i, just by counting the number of occurrences of the previous characters. If C is instead implemented as an array where the $C[c]$ holds the number of smaller characters than c in the input, finding the correct character can be done efficiently with a binary search. When we know which character is located at position i in the first column, we do $select_c(i - C[c])$ to determine where that occurence of c is located in the last column (the BWT), which is then returned as the result of the inverse LF function.

With the inverse LF function, computing the LCP value between two suffixes at position i and j is done by repeatedly applying the inverse LF function to i and j, and comparing the characters at those positions in the BWT.

TODO: Algorithm for iterating over suffixes which need to update their LCP value

TODO: Algorithm for updating a single LCP value

5.8 Dynamic clone extraction and source-mapping



The LCP is no longer represented as an array, but as a a balanced tree. As trees are more costly to traverse linearly compared to arrays, we also want to improve how we extract clones in this phase. The goal of this phase is still to find the indices in the fingerprint which corresponds to the start of code clones, which will later be mapped back to the original source code again.

Recall that the algorithm in the initial detection in chapter 4 traversed the suffixes from beginning to end, and extracts indices of suffixes where the LCP value is above the token threshold parameter. Also recall that some clones were filtered, those being clones which extend past a single fragment, or clones which are contained inside a larger clone. This algorithm now be recreated for the balanced tree LCP, which also optimizes the algorithm to reduce the amount of nodes we need to examine.

When updating a value in the LCP array, we can check to see if the new value is above the token threshold. If the previous value was below the treshold, and the new value is above the treshold, we know that this is a node which could potentially be a code clone, so storing these nodes can reduce the amount of nodes we need to examine. Since potentially multiple updates to an LCP value happens as potentially multiple edit operations are performed before extracting clone indices, we can reduce the amount of insertions/removals from this set by filtering nodes with a too low LCP value after we have done all LCP value updates.

After all LCP values are updated and we have filtered out nodes, we are left with a set of all

```
Algorithm DynamicCloneExtraction(nodesAboveTreshold)
    clones ← empty list
    lastAdded ← −1
    lastLCP ← −1
    for node in nodesAboveTreshold do
        index ← GetRankOfNode(node.pointer)
        if lastAdded ≠ −1 then
            difference ← index − lastAdded
            if node.key + difference ≤ lastLCP then
                continue
            end
        end
        Add(clones, node)
        lastAdded ← index
        lastLCP ← node.key
    end
    return clones
```

Algorithm 16: Extract clone indices in dynamic extended suffix array

the nodes with an LCP value above the token threshold parameter. In order to filter out clones which are contained within other clones, we need to add a similar check to the loop which was added in algorithm 8, however since the set only contains nodes with an LCP values above the token threshold, the algorithm will be slightly different. Instead of adding a loop which skips past indices which are contained clones, we can instead check if the last index added contains the current clone we are looking at. Doing the process in this fashion allows us to skip over many checks if there are long stretches of suffixes in the fingerprint with LCP values below the token treshold. Algorithm 16 shows how this algorithm is implemented.

After applying this algorithm we are left with the nodes with inorder ranks which correspond to the same indices as in the initial detection, and for the final phase, mapping the clone indices back to the original source code, we can do almost the same procedure as in the initial detection. The algorithm for this is almost the same as algorithm 10, but some parts are changed to avoid traversing through the trees many times to find the second index, which we would have to do if we used algorithm 10 without any modifications. The problem with algorithm 10 is that it computes the expression SA[ISA[firstIndex] − 1] to find the index of the matching clone. This is not a problem in the initial detection, because SA and ISA are arrays, which give $O(1)$ time access. This is not the case for the dynamic extended suffix array, which would require $O(\log n)$ access time for that expression. Instead, only the nodes of the clone indices in algorithm 16 is stored, and for a node $firstNode$, when we can find the node of the matching $secondNode$ in $O(1)$ time by finding the predecessor of firstNode and following its pointer. Remember that the nodes we have stored as clone indices are nodes in the A tree which corresponds to SA indices. Following the pointer corresponds to SA values, or alternatively, ISA indices. After we have both the firstNode and secondNode, we can get their fingerprint indices by getting the inorder rank of their pointer, with algorithm 14. Afterwards, the algorithm continues as previously. While this still requires traversing the tree in $O(\log n)$ time, we have reduced the number of traversals needed from 5 with a direct translation of 10 to 2 with the smarter traversal of nodes described here.

Now that we have successfully source-mapped from our dynamic extended suffix arrays to code clones, we are finished, and clones are sent to be displayed to the client.

Chapter 6

Evaluation

In this chapter, CCDetect-LSP will be evaluated based on different criteria, which combined will provide a basis for evaluating the tool as a whole.

First, we will verify that CCDetect-LSP correctly identifies type-1 clones. We will use the BigCloneBench [35] database and BigCloneEval [36] tool, which is the standard tools used to test clone detection accuracy.

Since the tool is focused on efficient detection of code clones, real-time performance of the tool will be a high priority in its evaluation. We will compare the time of the initial detection with the incremental updates. Note that we will also distinguish between the initial detection where parsing the entire code base is necessary, and subsequent detections which still constructs the suffix array from scratch, but does not require parsing the entire code base. We will call this type of detection the SACA detection, while the detection which uses the dynamic extended suffix arrays will be called the incremental detection. This is done to compare the dynamic extended suffix array against building the suffix array from scratch. Performance will be evaluated in two ways:

- Informal complexity analysis of phases in initial, SACA and incremental detection
- Performance benchmark of initial, SACA and incremental detection
- Performance benchmark comparison with iClones [14]

6.1 Verifying clones with BigCloneBench

In order to verify that CCDetect-LSP correctly identifies clones, we should analyze and confirm that CCDetect-LSP finds all clones in a given code base. BigCloneBench [35] is a clone detection benchmark of a set of known clones in a dataset called "IJaDataset". The benchmark consists of the IJaDataset and a database containing information of the clones which exist in the dataset. BigCloneEval [36] is a tool which simplifies evaluation of a tool on the BigCloneBench. BigCloneEval evaluates a tool by running the tool on the dataset, letting the tool output all the clones the tool finds, and then matching the clones the tool found with the clones in the

database. The clones in the database are manually verified by humans, and include type-1 to type-3 clones.

We evaluated CCDetect-LSP by running the initial detection algorithm on the dataset, and outputting all the clones the tool found. BigCloneEval expects to get a file where each line contains a clone pair, where a clone is specified with filename, starting and ending line. This was simple to extract from our clone-map, where we converted our ranges which point to the beginning and ending token, to just the line number of those tokens.

BigCloneEval reports the recall of a tool, meaning the percentage of found clones. Therefore, we did not make sure that our conversion to the format BigCloneEval expects gave the minimum number of clone pairs. For example, for a clone pair of clones A and B, we output both that A is a clone of B and that B is a clone of A, which is superfluous. This would lead to a bad precision, but does not affect the recall in the report which BigCloneEval outputs.

We used the following command and parameters to evaluate CCDetect-LSP:

./evaluateTool –min-tokens=100 -s=100 –output=ccdetect.report -t=1

For CCDetect-LSP, we used mostly the default parameters of BigCloneEval, but we increased the minimum clone size to 100 and set the minimum similarity threshold to be 100%, since we are only evaluating CCDetect-LSP for type-1 clone recall. This was done to reduce the time an evaluation takes as it does not consider type-3 clones when the similarity threshold is set to 100%, and the default token threshold of 50 requires a lot of time to match clones with the database.

Appendix 9.1 shows the beginning of the report generated by BigCloneEval when CCDetect-LSP is evaluated. The report shows that CCDetect-LSP has a 99.98% recall for type-1 clones. CCDetect-LSP is clearly capable of detecting type-1 clones. As for the missing type-1 clones, while it is not so simple to determine which clones were not detected, it is possible that these clones are not reported because of inconsistencies in the database. The report also shows that CCDetect-LSP manages to detect a few type-2 clones. Detecting these are accidental, as CCDetect-LSP does not currently implement any normalization of the input which would allow for type-2 detection. These type-2 clones are likely detected because BigCloneEval allows some leniency in terms of how much a clone reported by CCDetect-LSP needs to match with a clone reported in the BigCloneBench database. Therefore, it is likely the case that a few type-2 clones overlap with a type-1 clone, and BigCloneEval matched these clones because of its leniency.

6.2 Time complexity of detection

In this section we will conduct an informal analysis of the running time of each phase of the initial, SACA and incremental detection to argue that the average runtime complexity of the incremental detection is more efficient than the initial detection. In each phase we will argue the run time in terms of Big O notation. Some claims will be substantiated in the next section where we look at concrete code bases and their properties.

The initial detection runs in $O(n)$ time, where n is the number of characters in the code base. The bottleneck of the initial detection is reading and parsing all the content in each file. Tree-sitter generates Generalized LR (GLR) parsers [24], which in the worst-case have a $O(n^3)$ running

time, but $O(n)$ for any deterministic grammar. As programming languages generally have deterministic grammars, we will assume that the running time of parsing with Tree-sitter takes $O(n)$ time. After the initial parsing, the SACA detection runs in $O(f)$ time, where f is the size of the fingerprint and $f \ll n$. The running time is $O(f)$ because the suffix array construction is performed for every update, which takes linear time in the size of the input, which is the fingerprint. The extraction of clones from the LCP array also runs in $O(f)$ as it is a single scan over the LCP array. The final source-mapping is a bit more complicated, taking $O(|\text{clones}| \times \log(|\text{documents}|))$. This complexity comes from binary-searching the list of documents to find the correct document for each clone. This is highly likely to be less time consuming than the suffix array construction, as the number of documents and number of clones are usually orders of magnitude lower than the size of the whole code base. Therefore, we get a final running time of $O(f) + O(|\text{clones}| \times \log(|\text{document}|))$, where $O(f)$ is highly likely to be the slowest factor.

For the incremental detection, we have already parsed the code base and built the index and dynamic extended suffix array structure for the code base. Afterwards, when an edit E is performed in a document D, the first phase is to update the document index. We iterate over the documents to update their fingerprint ranges, and when a document which has been edited is reached, the new document content is incrementally re-parsed, queried for fragments and then the fragments are fingerprinted. In the worst-case, we have to fingerprint the entire document, which has a complexity of $O(|D|)$.

Next, extracting the edit operations which have happened to D_f (fingerprint of D), takes $O(|D_f| + |E|^2)$ where $|E| \leq |D_f|$. Note that the size of the edit is calculated as the area which E covers, meaning that if an edit consists of changing a token at the beginning of the file, and a token at the end of the file, then $|E| \approx |D_f|$. We get this time complexity because Hirschberg's algorithm runs in $O(n \times m)$ where in our case, $n \approx m$. If the size of the edit is contained in a smaller area, we apply the optimization which reduces the size of the edit by comparing characters at the beginning and end of the string, as discussed in chapter 5. This process has a complexity of $O(|D|)$, and afterwards Hirschberg's algorithm has a worst-case complexity of $O(|E|^2)$.

The worst-case complexity of dynamically updating the extended suffix array is actually slower than a linear time SACA algorithm in the worst-case. The worst-case scenario when inserting/deleting a character in the fingerprint is that every single suffix needs to be reordered, meaning we reorder $O(f)$ suffixes, where each reordering takes $O(\log(f))$ time, as it requires deleting and inserting an element in the dynamic extended suffix array. In addition, we need to call the LF function twice for each reordering, which has a $O(\sigma + \log f \log \sigma)$ complexity. The complexity of the LF function comes from a *rank* call in the wavelet matrix, and an iteration over C to find the number of smaller characters. This results in a $O(f \times (\log f + \sigma + \log f \log \sigma))$ running time of this phase, which is worse than the $O(f)$ running time of the SACA algorithm. In addition, before the reordering when we are inserting/deleting characters in the BWT, we use the LF function and insert/delete in the wavelet matrix and the C array. The highest complexity of these operations is again the LF function. If an edit operation consists of inserting/deleting multiple characters, the number of LF function calls is increased, with a complexity of $O(e \times (\sigma + \log f \log \sigma))$, where e is the number of characters inserted/deleted. Note that while there are many factors in this time complexity, all the factors should be quite small compared to f, making the complexity of reordering the suffixes the slowest factor of this phase.

The average-case complexity of this phase is however highly likely to be faster. Salson et al. [25] have shown that on average, the number of reorderings required for an insertion/deletion in a

suffix array is highly correlated with the average LCP value of the input. Their data shows that for multiple different types of data such as genome sequences and english text, the average LCP value of the input is generally magnitudes lower than the input size. In our experiments, this applies to source code as well, as code bases we have tested on have all had an average LCP values well below 100. See table 6.1 to see the average LCP value for different codebases. A lower number of reorderings for lower LCP values seems intuitive, as lower LCP values mean that the insertion/deletion will affect the ordering of fewer suffixes in the input. With this information, it would be more accurate to downplay the importance of the $O(f)$ number of reorderings in our analysis, and we therefore claim that the average running time of an edit operation on the suffix array is closer to $O(\log f + (e \times \sigma + \log f \log \sigma))$. We extend this to account for multiple edit operations as well, for each edit operations which was computed in the last phase, we perform an insertion/deletion of e characters. Therefore, the total complexity of this phase on average is closer to $O(|\text{edits}| \times (\log f + (e \times \sigma + \log f \log \sigma)))$.

Next there is the clone extraction phase. Recall that in the dynamic detection clone extraction phase, we had stored all nodes with an LCP value above the token threshold, and iterate over those to determine which of them are clones or not. In the worst-case, every index in the LCP array would be above the token threshold, which would be an $O(f)$ complexity iteration. However, this is highly unlikely, since the nodes with LCP value above the token threshold are either clones, or contained clones. The number of contained clones is limited by the number of clones because each clone can only have a limited amount of contained clones. Therefore, the number of LCP nodes examined should be closer to $O(|\text{clones}|)$, which is highly likely to be much less expensive than iterating over all the LCP nodes. For each examined LCP node, we need to traverse from the node, to its pointer node, and then to the root of the B tree to determine the fingerprint index of that LCP node. We do the same to determine the index of the matching clone, as described in chapter 5. These traversals take $O(\log f)$ time. The worst-case performance of this phase is therefore $O(f \times \log(f))$, but we will see that the average-case performance is closer to $O(|\text{clones}| \times \log(f))$.

Finally, the source-mapping phase is easier to analyze. As we now know all the positions of clones and their matches, building the clone-map takes $O(|\text{clones}| \times \log(|\text{documents}|))$ time. We perform the binary-search over the documents to get the source location for each clone index we found in the previous phase. This is the same complexity for both the SACA and incremental detection.

With this informal analysis, we have shown that in the average case, the incremental detection should be faster than the SACA detection, as none of the phases reach or exceed the $O(f)$ running time of the SACA detection. In the next section we will show that the properties we have assumed for this analysis are present in multiple code bases, and that these properties do lead to better performance for the incremental detection.

6.3 Benchmark performance

For the performance benchmark we will benchmark the running time of CCDetect-LSP on code bases and edits of different sizes. We will compare the performance of the incremental detection with the SACA detection and iClones [14].

iClones is a tool which similarly to CCDetect-LSP, does incremental clone detection. The tool takes n revisions of the same code base and will after the initial detection, reuse as much infor-

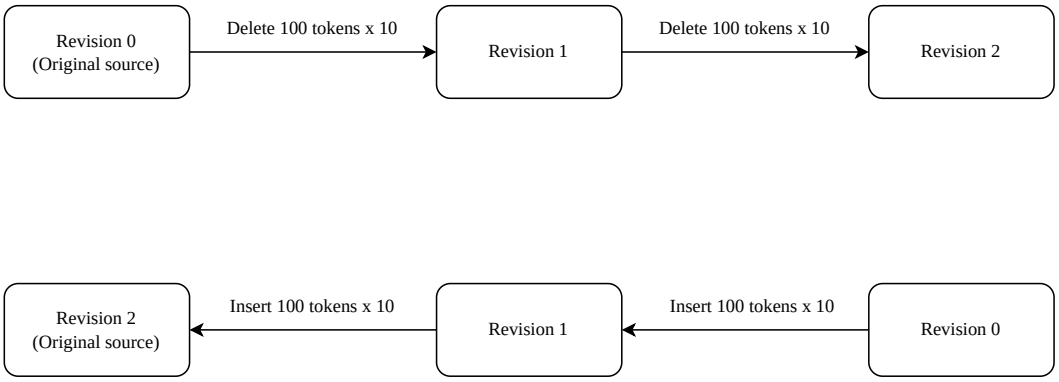


Figure 6.1: Reversion of an evaluation test. Delete100x10 to Insert100x10

mation as possible to reduce the amount of work needed to analyze consecutive revisions. The algorithm which iClones uses is based on generalized suffix trees (GST), and the tool can detect up to type-3 clones. For a revision i, iClones excepts to have a file named `changed` in the root folder of the revision, which contains information on which files have changed between revision $i-1$ and i. For each file which has changed, the suffixes in the file is inserted into the GST, reusing nodes of the tree if possible.

For the performance benchmark, the code bases are set up in the way iClones expects. For each code base to analyze, we will store n revisions of it, where each revision contains some changes to some of the files. An evaluation program for CCDetect-LSP was written so that it can process this same setup, making it simple to run both CCDetect-LSP and iClones on the same code bases, and compare their results.

For each code base, we can generate new revisions. Generating new revisions of a code base means that we copy the code base and apply a few insertions/deletions to random files, to simulate how a programmer would edit files when programming. Generating new revisions of code bases is not a trivial problem, since each edit performed in a file should still give a valid program which can be parsed. If the file cannot be parsed, Tree-sitter will not give a proper AST for the file. Because of this, generating insertions is hard, because we need to make sure that the location we are inserting some code is a valid location for that code. It is easier to generate deletions first instead. Deletions are easier to generate, as we can use Tree-sitter to determine where a fragment of size n is located, and remove that section of code in the next revision. For example in Java, we can delete an entire method, and still know that the program will parse to a valid AST afterwards. After we have generated for example 10 revisions where the difference between each revision is deletion of some number of fragments, we can then reverse the ordering of the revisions, so that the final revision is now the first revision, and so on. This will translate each deletion to instead be an insertion, as an insertion is simply an inversion of a deletion. Figure 6.1 show graphically how a test consisting of three revisions with deletions can be inverted to create a test of insertions.

With this technique, generating an arbitrary amount of tests is now possible. There are multiple variables we can tune when generating the tests. Primarily the three most important factors of the tests is the size of the code base, the number of edit operations in each revision, and the size of those edit operations.

Code base	LOC	Clones detected	LCP_{avg}	$\text{LCP}_{\geq 100}$
WorldWind	550KLOC	1517	18	63967
neo4j	1MLOC	1313	9	27557
graal	2.2MLOC	2012	28	154452
flink	2.3MLOC	4729	13	155754
elasticsearch	3.2MLOC	9986	14	289511
intellij-community	5.8MLOC	3585	19	336190

Table 6.1: Properties of code bases

The size of the code base will naturally affect the running time of the algorithm. We randomly picked open-source code bases of differing sizes and differing amounts of duplication to run the benchmark on. We selected the code bases WorldWind[1], neo4j[2], graal[3], flink[4], elasticsearch[5] and intellij-community[6]. The code bases chosen were different size Java code bases, with different amounts of duplication. Table 6.1 shows some properties of the code bases, including its number of lines, the number of clones detected by CCDetect-LSP for a token threshold of 100, the average LCP value and the number of LCP values above 100. An interesting aspect of the graal and flink code bases is that they both contain $\sim$ 2.2MLOC, but graal has a much higher LCP_{avg}, which could potentially lead to a slower suffix array update.

Additionally, we chose to test both an edit size of 10 and 100 with 10 edits in each revision. We believe these values to be slightly larger than what is realistic for an IDE scenario where a programmer is editing and updating the fingerprint of a single file at a time. For a test where we are performing 10 insertions of 100 tokens, this means we are inserting 1000 tokens in a single edit, which is likely larger than any realistic edit. Therefore, if the data shows that the algorithm can handle these types of edits in "real-time", we can claim that the algorithm can handle most realistic editing scenarios.

The benchmarks will be performed on a computer with an Intel i7-2600K CPU with a 3.4GHz clock speed and 4 cores, and 16GB RAM. The SACA detection, incremental detection and iClones will be run with a token threshold of 100, and the processing time of each revision will be timed using a simple clock mechanism programmed in Java.

TODO: Average results over multiple runs

TODO: A single run over a large code base showing what happens when number of edit operations increase

Figure 6.2, 6.3,6.4, 6.5, 6.6 and 6.7 shows the benchmark results on the different code bases. Each graph shows the running time of the SACA detection, the incremental detection and iClones incremental detection. The running time is shown on a log scale, as the initial detection is extremely slow compared to the subsequent detections. iClones was not able to run on the intellij-community code base, as the memory usage exceeded 16GB, therefore only the CCDetect-LSP algorithms is shown on those graphs. The elasticsearch code base with a size of 3.2MLOC was

[1]WorldWind: `https://github.com/NASAWorldWind/WorldWindJava`
[2]neo4j: `https://github.com/neo4j/neo4j`
[3]graal: `https://github.com/oracle/graal`
[4]flink: `https://github.com/apache/flink`
[5]elasticsearch: `https://github.com/elastic/elasticsearch`
[6]intellij-community: `https://github.com/JetBrains/intellij-community`

approximately the maximum size code base iClones could handle memory wise.

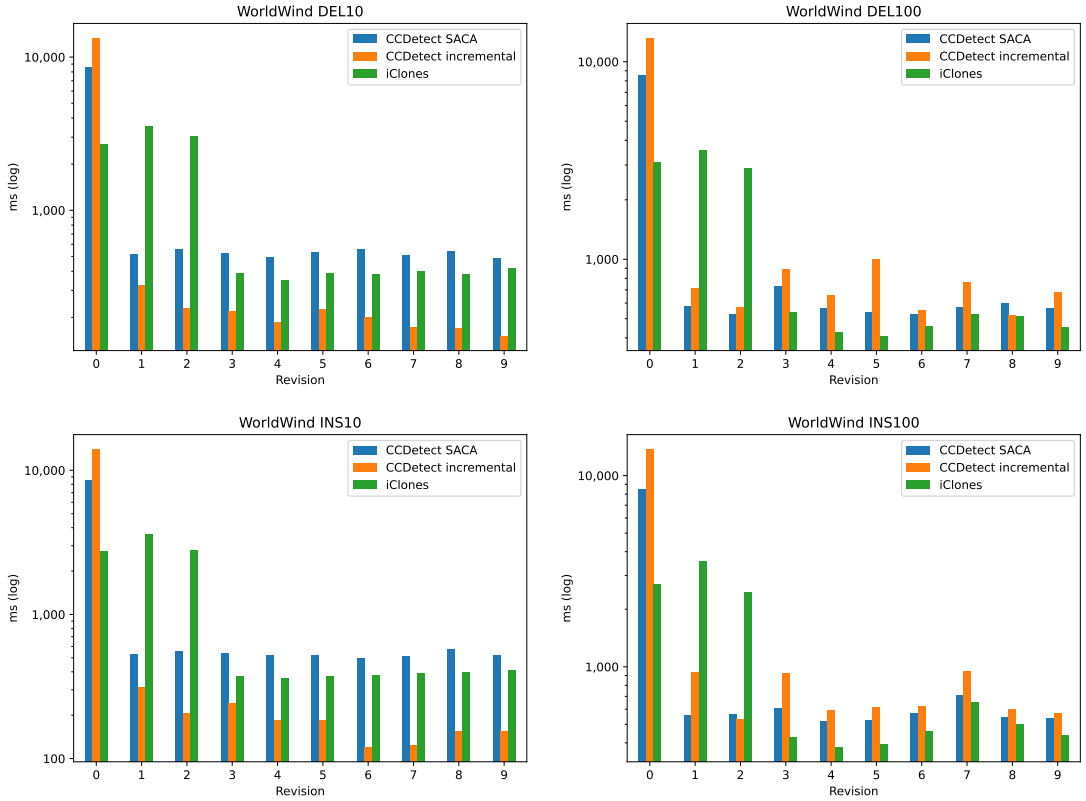


Figure 6.2: WorldWind performance benchmark

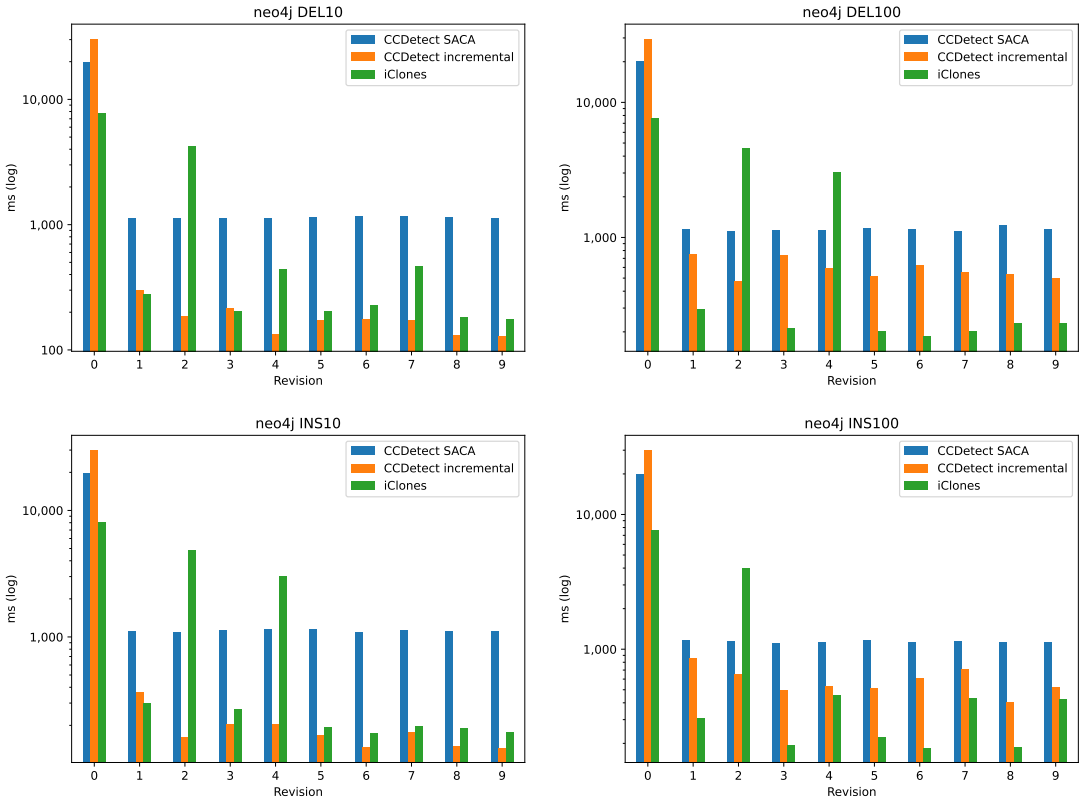


Figure 6.3: neo4j performance benchmark

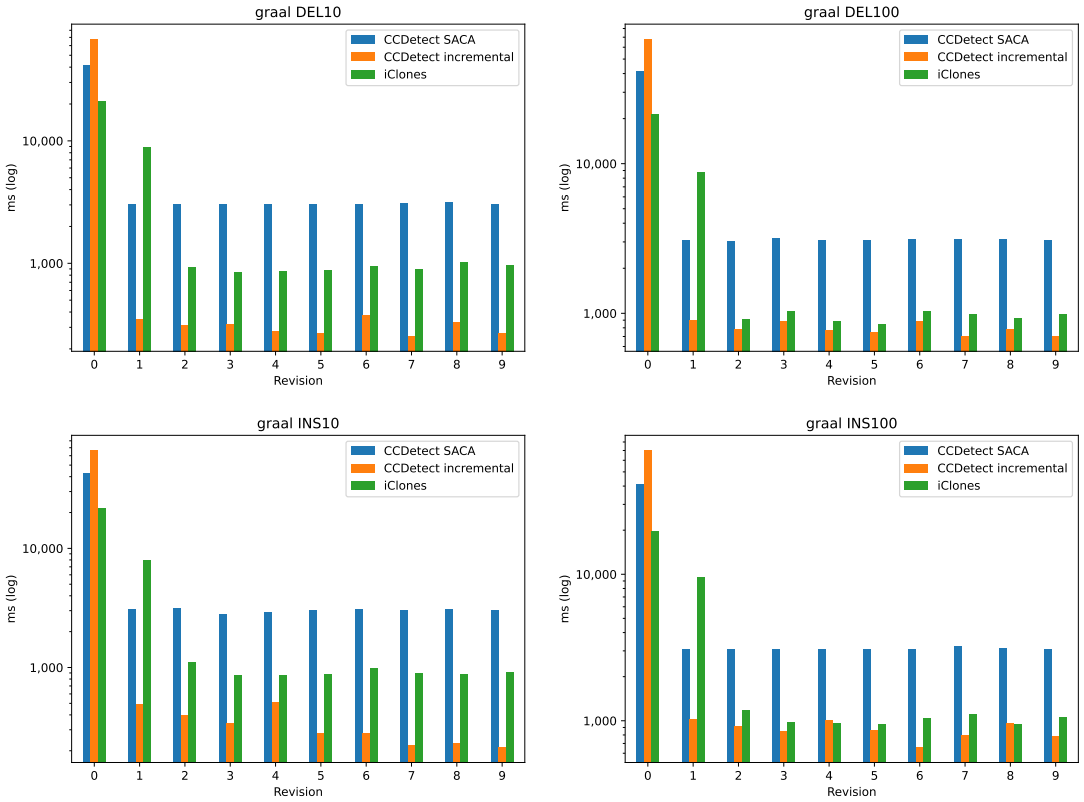


Figure 6.4: graal performance benchmark

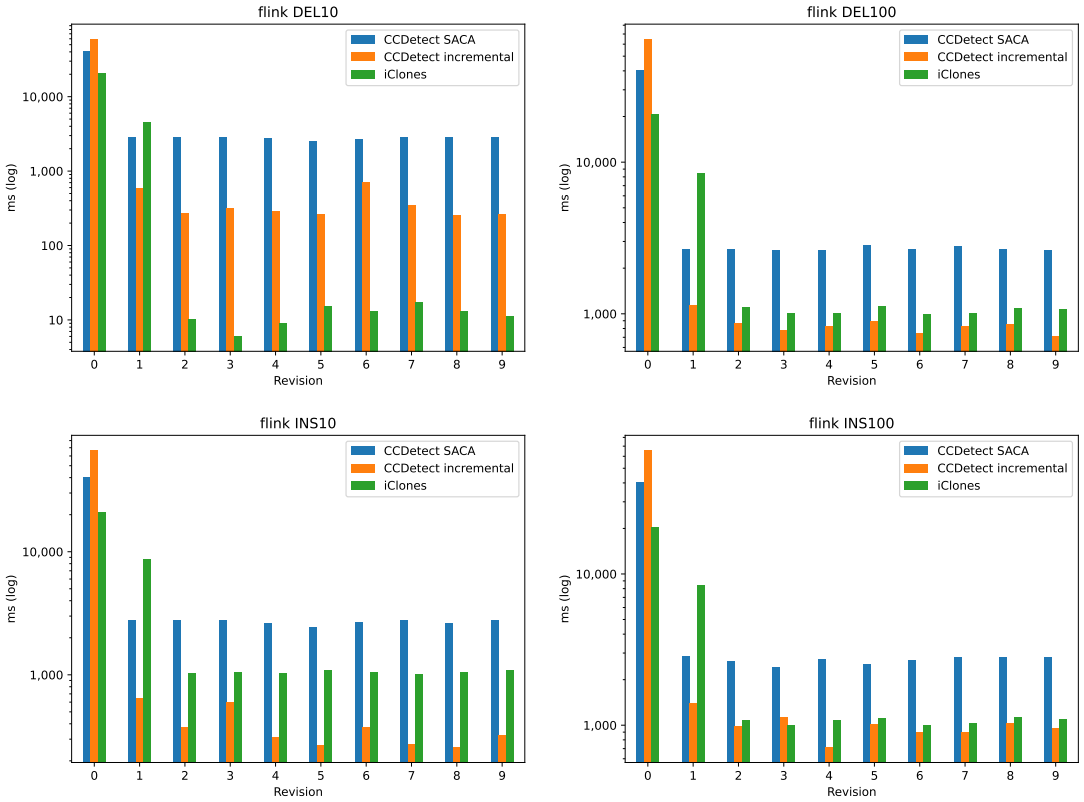


Figure 6.5: flink performance benchmark

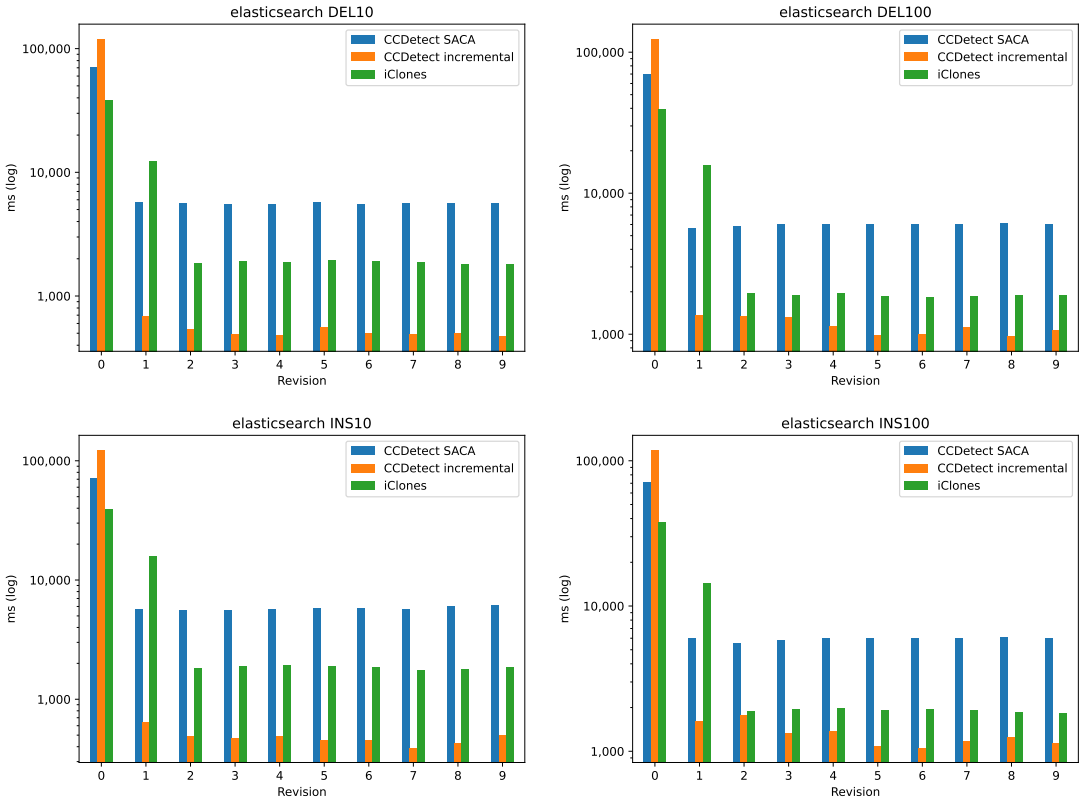


Figure 6.6: elasticsearch performance benchmark

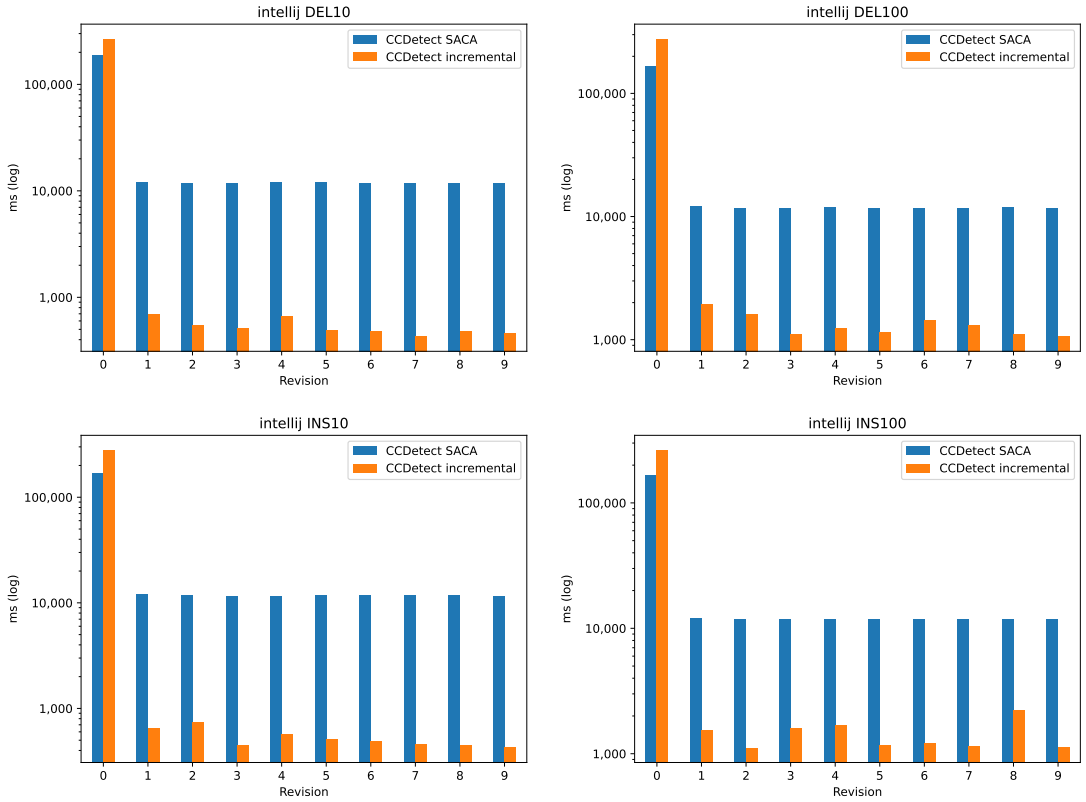


Figure 6.7: intellij-community performance benchmark

6.4 Memory usage

Chapter 7

Discussion

7.1 Performance

7.2 Clones

Chapter 8

Conclusion and future work

8.1 Future work

In this section we list future work and research in order of what we consider most interesting to do more research on, to least.

Optimal edit operations

Refactoring of clones

Compressing data structures

Lazy LCP array updates

8.2 Related work

8.3 Conclusion

Chapter 9

Appendix

```
Algorithm WagnerFischerOperations(S1, S2, matrix)
    operations ← Empty list
    currentOperation ← None operation
    lastOperationIndex ← −1
    x ← Len(matrix) − 1
    y ← Len(matrix[x]) − 1

    while x > 0 and y > 0 do
        subValue ← matrix[x − 1][y − 1]
        delValue ← matrix[x − 1][y]
        insValue ← matrix[x][y − 1]
        if subValue ≤ delValue and subValue ≤ insValue and  then
            if subValue ≠ matrix[x][y] then
                if currentOperation is not a substitute or lastOperationIndex ≠ x then
                    currentOperation ← New substitute operation with position x − 1
                    Insert(operations, 0, currentOperation) // Add operation at index 0
                end
                else
                    currentOperation.position ← currentOperation.position − 1
                end
            end
            Add S2[y − 1] to beginning of currentOperation.chars
            lastOperationIndex ← x − 1
            x ← x − 1
            y ← y − 1
        end
        else if insValue ≤ delValue then
            if currentOperation is not an insert or lastOperationIndex ≠ x then
                currentOperation ← New delete operation with position y − 1
                Insert(operations, 0, currentOperation) // Add operation at index 0
            end
            Add S1[x − 1] to beginning of currentOperation.chars
            lastOperationIndex ← x − 1
            y ← y − 1
        end
        else
            if currentOperation is not a delete or lastOperationIndex ≠ x then
                currentOperation ← New insert operation with position x − 1
                Insert(operations, 0, currentOperation) // Add operation at index 0
            end
            Add S1[x − 1] to beginning of currentOperation.chars
            lastOperationIndex ← x − 1
            x ← x − 1
        end
    end
    return operations
```

Algorithm 17: Compute edit operations from Wagner-Fischer matrix

```
Algorithm WagnerFischerEditDistance(S_1, S_2)
    n ← Len(S_1)
    m ← Len(S_2)
    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 S_1[i − 1] = S_2[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 18: Fill edit distance matrix using Wagner-Fischer algorithm

```
-- Versioning --
 BigCloneEval: Release Version 0.1
BigCloneBench: Version 1.0 (2016-06-19)
-- Selected Clones --
         Min Lines: null
         Max Lines: null
        Min Tokens: 100
        Max Tokens: null
  Min Pretty Lines: null
  Max Pretty Lines: null
       Min Judges: null
    Min Confidence: null
          Sim Type: BOTH
Minimum Similarity:

-- Clone Matcher --
Coverage Matcher.  Coverage Ratio = 0.7, minimum raito is of reference clone: null

================================================================================
All Functionalities
================================================================================
-- Recall Per Clone Type (type: numDetected / numClones = recall) --
              Type-1: 23205 / 23210 = 0.9997845756139595
              Type-2: 55 / 3547 = 0.015506061460389062
      Type-2 (blind): 2 / 245 = 0.00816326530612245
 Type-2 (consistent): 53 / 3302 = 0.016050878255602665

-- Inter-Project Recall Per Clone Type (type: numDetected / numClones = recall)  --
              Type-1: 4620 / 4625 = 0.9989189189189189
              Type-2: 24 / 3185 = 0.0075353218210361065
      Type-2 (blind): 0 / 223 = 0.0
 Type-2 (consistent): 24 / 2962 = 0.008102633355840648

-- Intra-Project Recall Per Clone Type (type: numDetected / numClones = recall) --
-- Recall Per Clone Type --
              Type-1: 18585 / 18585 = 1.0
              Type-2: 31 / 362 = 0.0856353591160221
      Type-2 (blind): 2 / 22 = 0.09090909090909091
 Type-2 (consistent): 29 / 340 = 0.08529411764705883
```

Figure 9.1: BigCloneEval evaluation report on CCDetect-LSP

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] Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez Pereira. "The wavelet matrix: An efficient wavelet tree for large alphabets". In: *Inf. Syst.* 47 (2015), pp. 15–32. DOI: 10.1016/j.is.2014.06.002. URL: https://doi.org/10.1016/j.is.2014.06.002.

[10] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[11] 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.

[12] 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.

[13] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley Professional, 1994. ISBN: 0201633612. URL: http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1.

[14] 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.

[15] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. "High-order entropy-compressed text indexes". In: *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA*. ACM/SIAM, 2003, pp. 841–850. URL: http://dl.acm.org/citation.cfm?id=644108.644250.

[16] D. S. Hirschberg. "A Linear Space Algorithm for Computing Maximal Common Subsequences". In: *Commun. ACM* 18.6 (June 1975), pp. 341–343. ISSN: 0001-0782. DOI: 10.1145/360825.360861. URL: https://doi.org/10.1145/360825.360861.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] Bernard Lang. "Deterministic Techniques for Efficient Non-Deterministic Parsers". In: *Automata, Languages and Programming*. Ed. by Jacques Loeckx. Berlin, Heidelberg: Springer Berlin Heidelberg, 1974, pp. 255–269. ISBN: 978-3-662-21545-6.

[25] M. Léonard, L. Mouchard, and M. Salson. "On the number of elements to reorder when updating a suffix array". In: *Journal of Discrete Algorithms* 11 (2012). Special issue on Stringology, Bioinformatics and Algorithms, pp. 87–99. ISSN: 1570-8667. DOI: https://doi.org/10.1016/j.jda.2011.01.002. URL: https://www.sciencedirect.com/science/article/pii/S1570866711000037.

[26] Veli Mäkinen and Gonzalo Navarro. "Rank and select revisited and extended". In: *Theoretical Computer Science* 387.3 (2007). The Burrows-Wheeler Transform, pp. 332–347. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2007.07.013. URL: https://www.sciencedirect.com/science/article/pii/S0304397507005300.

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] M. Salson et al. "A four-stage algorithm for updating a Burrows–Wheeler transform". In: *Theoretical Computer Science* 410.43 (2009). String Algorithmics: Dedicated to Professor Maxime Crochemore on the occasion of his 60th birthday, pp. 4350–4359. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2009.07.016. URL: https://www.sciencedirect.com/science/article/pii/S0304397509004770.

[34] 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.

[35] 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.

[36] Jeffrey Svajlenko and Chanchal K. Roy. "BigCloneEval: A Clone Detection Tool Evaluation Framework with BigCloneBench". In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2016, pp. 596–600. DOI: 10.1109/ICSME.2016.62.

[37] 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.

[38] Esko Ukkonen. "On approximate string matching". In: *Foundations of Computation Theory*. Ed. by Marek Karpinski. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 487–495. ISBN: 978-3-540-38682-7.

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.