SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Code Clone Detection in Isabelle Using Token Stream Similarity

Seifeddine Ghanouchi

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Code Clone Detection in Isabelle Using Token Stream Similarity

Erkennung von Klonen in Isabelle Code durch Token-Stream Ähnlichkeit

Author: Seifeddine Ghanouchi
Supervisor: Prof. Tobias Nipkow
Advisor: Fabian Huch
Submission Date: 22.05.2023

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and materials used.

Munich, 22.05.2023 Seifeddine Ghanouchi

Acknowledgments

I would like to first thank my advisor Fabian Huch for his help ranging from explaining the core Isabelle concepts as well as in the decision making process and providing constructive feedback throughout the span of this thesis. I would also like to thank my partner, friends and family for their constant support and help.

Abstract

Rewriting or even copying and pasting chunks of code is commonly done by most, if not all developers. However, having code clones inside projects can be problematic, as it can lead to errors or a plain increase in the workload. This can be further extended to theorem provers, such as Isabelle.
A clone detection tool was implemented previously using a token-based approach. This approach works by looking for the longest common sequence of tokens and only detects segments of copy-pasted code. However, the clone detection scheme should be able to detect similar blocks, despite small changes in individual tokens. Therefore, a fuzzy token matching algorithm is used to identify blocks of code with similar token streams. This approach works by performing a maximum weight matching between two lists of tokens and filtering token pairs with a similarity value lower than a chosen threshold δ. Code blocks are considered clones if their similarity value is above a chosen threshold Δ. The algorithm is further optimized through the use of filtering steps to avoid unnecessary costly comparisons: we propose a signature scheme to create a set of code clone candidates. Then we filter the candidates using a weight condition and a relaxed matching. We conclude that the resulting clones for $\delta = 0.85$ and $\Delta = 0.8$ are copy-paste as well as renamed clones. Furthermore, code blocks with a similarity greater than 0.5 contain similar parts and statements but are not clones. We also observe that the filtering steps greatly improve the speed of the algorithm.

Contents

Acknowledgments iii

Abstract iv

1 Introduction 1

2 Theoretical Background 2
2.1 Code Clones 2
2.2 String Similarity Approaches 4
2.2.1 Character-Based Approach 4
2.2.2 Token-Based Approach 4
2.2.3 Hybrid-Based Approach 5
2.3 Fuzzy Token Matching 6
2.4 Signatures in Fuzzy Token Matching 6
2.5 Isabelle Code Structure 7

3 Related Work 8
3.1 Clone Detection Schemes for Isabelle Code 8
3.1.1 Token-Based Clone Detection Using Suffix Trees in Isabelle 8
3.1.2 Clone Detection on Isabelle Terms based on Abstract Syntax Trees 8
3.2 String Matching Approaches 8
3.2.1 Running Karp-Rabin Greedy String-Tiling Algorithm 8
3.2.2 Fast-join: An efficient method for fuzzy token matching based string similarity join 9

4 Fuzzy Token Matching in Isabelle 10
4.1 Token Markup 10
4.2 Token Comparison Scheme 11
4.3 Token Matching 12
4.4 Optimizations 14
4.4.1 Signature Schemes 14
4.4.2 Weight Condition 16
4.4.3 Relaxed Matching 17

4.5 Summary 18

5 Result Evaluation **20**
5.1 Analysis of Sample Results 20
5.2 Analysis of the Filtering Steps 21
5.2.1 Analysis of the pruned possibilities 22
5.2.2 Influence on the speed of the clone detection 22

6 Conclusion **25**

7 Future Work **26**
7.1 Improvements to the Proposed Algorithm 26
7.2 Outlook and Further Features 27

List of Figures **28**

List of Tables **29**

Bibliography **30**

1 Introduction

When we need to reuse a functionality that has been already implemented, whether online or in a local project, a common practice is to just copy and paste the given code. It is also common to reuse the same chunk of code in more than one location. There are multiple reasons behind this behavior. These reasons include the following [RBS13]:

- **Time constraints**: programmers need to finish a task as fast as they can.
- **Complex project**: if programmers do not understand the structure of a system, they tend to copy existing functionality and logic instead of writing their own.
- **Programming language issues**: if a programmer is not very familiar with a specific programming language, they often end up copying and reusing code chunks instead of writing their own.

However, these issues lead to the presence of code clones in the project. Having multiple clones means that whenever that specific part needs to be updated, each clone of the code needs to be individually updated, therefore increasing the workload [RC07]. Additionally, it is common to forget to update one of the clones or more, which leads to errors or inconsistent behavior. Therefore, the presence of code clones should be avoided as much as possible.

Isabelle is a generic proof assistant. It is used to express mathematical formulas in a formal language and to prove these formulas in a logical calculus. Isabelle comes with a large theory library of formally verified mathematics[1]. Therefore, the topic of code clone detection is relevant to Isabelle as well. In this work, we propose an algorithm to detect code clones or code blocks with a high degree of similarity.

[1]https://isabelle.in.tum.de/overview.html

2 Theoretical Background

In this chapter, the necessary theoretical background is presented. First, we present a few code clone types. Then we discuss different metrics and methods for how they can be identified. Furthermore, we present the necessary context for Isabelle.

2.1 Code Clones

As stated by Qurat Ul Ain et al., there is no formal definition of a code clone. They can be chunks of similar code as well as blocks of identical code. Code clones are categorized using these four basic types [Ain+19]:

- **Exact clones (Type 1)**: Code segments that are identical, excluding comments, line breaks and white spaces.
- **Renamed clones (Type 2)**: Code segments that are identical syntactically or structurally but have different identifiers, types, literals and layouts.
- **Near miss clones (Type 3)**: Code segments with a similar structure and syntax but with some statements being added, deleted or modified.
- **Semantic clones (Type 4)**: Code segments that are written differently but are functionally similar.

Figure 2.2 shows examples of the different code clone types based on the pseudo-code in Figure 2.1. A clone of type 1 is represented in (a). The only difference compared to the base code is a line break. An example of type 2 is written in (b). This can be seen through the renamed function and variable from `hello_world` to `not_hello_world` and from `result` to `hello`. Subfigure (c) shows a type 3 clone by omitting a statement. A clone of type 4 is represented in (d). This can be seen by the different structure to accomplish the same task: instead of directly printing the string, each character is individually printed. Clones of types 1, 2 and 3 have a lot of textual similarities. Therefore, they can be detected using different string similarity algorithms. However, the detection of type 4 clones requires semantic approaches, where the logic and functionality of the different code blocks is compared. The focus of the work is detecting types 1 and 2, as well as some type 3 code clones.

```
void hello_world(){
 String result = "hello world!";
 println(result);
}
```

Figure 2.1: Example base pseudo-code

```
void hello_world()
{
 String result="hello world!";
 println(result);
}
```

(a)

```
void not_hello_world() {
 String hello = "hello world!";
 println(hello);
}
```

(b)

```
void not_hello_world() {
 println("hello world!");
}
```

(c)

```
void not_hello_world() {
 String result = "hello world!";
 for (int i = 0; i < result.length; i++) {
    print(result.charAt(i));
 }
 println();
}
```

(d)

Figure 2.2: Example code clones of the four main types

2.2 String Similarity Approaches

String matching tools are a reliable solution to detecting code clones of types 1, 2 and 3 [Ain+19]. The following sections present the main three approaches to string matching.

2.2.1 Character-Based Approach

The similarity between two given strings is measured using the individual characters of the strings. A common metric to measure the string similarity in a character-based approach is the edit distance (ED). ED of the strings S and S' is the minimum number of single-character edit operations to transform S into S' or vice versa. A single-character edit operation is one of three possibilities: it is either the addition, the deletion or the modification of a selected character. The ED can be measured using multiple algorithms, among them is the Levenshtein distance [GF13]. For example, ED(“bachelor thesis”, “bachelor theses”) $= 1$, whereas ED(“string match”,“match string”) $= 12$. These examples show that changing individual characters in words without altering their order keeps the ED relatively small, indicating a great similarity between strings. However, changing the order of words leads to a noticeably higher ED, despite the sentences being similar. Another disadvantage of the ED is the difficulty of interpreting its results: the same ED value may indicate that two short strings are entirely different but also that two long strings are very similar.

The normaized ED (NED) is a workaround for this last disadvantage: $NED(S, S') \in [0, 1]$ with 1 or 0 indicating that S and S' are identical or completely different, respectively. NED is commonly referred to as edit similarity and is calculated with the following formula:

$$NED(S, S') = 1 - \frac{ED(S, S')}{max(|S|, |S'|)}$$

Despite the NED solving the problem of interpreting the results, it still fails at assessing the similarity of strings with the same words in different order.[WLF11]

2.2.2 Token-Based Approach

This approach consists in dividing each given string into a set of substrings, referred to as *tokens*. The token set of a string is created following a tokenization scheme, for example dividing the string into different words based off of whitespaces. For two given strings S and S' with their respective token sets T and T', the similarity is calculated based on $T \cap T'$.

The following formulas are some metrics for measuring the token-based similarity [GF13]:

- **Dice Similarity:** $DICE(S,S') = \frac{2.|T \cap T'|}{|T|+|T'|}$
- **Cosine Similarity:** $COSINE(S,S') = \frac{|T \cap T'|}{\sqrt{|T|+|T'|}}$
- **Jaccard Similarity:** $JACCARD(S,S') = \frac{|T \cap T'|}{|T|+|T'|-|T \cap T'|}$

An advantage that token-based approaches offer compared to character-based ones is the following: when using tokens, the order of different words does not affect the similarity. For example, the two strings $S =$ "match string" and $S' =$ "string match" with their respective token sets $T = \{$"match","string"$\}$ and $T' = \{$string","match$\}$ have the following properties:

$$T = T' = T \cap T'$$

$$DICE(S,S') = COSINE(S,S') = JACCARD(S,S') = 1$$

These results indicate that the token sets are considered identical and hence the strings similar.
The main disadvantage in this approach is that $T \cap T'$ only includes tokens that are present in both T and T'. This means that tokens that are similar but not identical are not in $T \cap T'$. For instance, the strings $S =$ "bachelor thesis" and $S' =$ "Bachelor Thesis" with token sets $T = \{$"bachelor", "thesis"$\}$ and $T' = \{$"Bachelor", "Thesis"$\}$ have a similarity of 0, despite the only difference being two letters.[WLF11]

2.2.3 Hybrid-Based Approach

Hybrid approaches combine the two previous approaches to compensate for the disadvantages of both. This method extends the character-level operator on the token-level. This means that each pair of tokens will be compared using the character-based approach. An example usage of the hybrid approach is the approximation of generalized edit similarity ($AGES$) [WLF11]. This metric iterates through each token and finds the token in the other set with the highest edit similarity and saves this value. The $AGES$ is the average of the saved edit similarities. This method does not follow the symmetry property, making it inconsistent in judging the similarity between two texts. For example, the two strings $S =$"string search" and $S' =$"string matching research" get divided into the token sets $T = \{$" string "," search"$\}$ and $T' = \{$"string", "matching", "research"$\}$ respectively. To calculate $AGES(S,S')$, the tokens in T get the following assignments: ("string", "string") and ("search", "research"). Therefore $AGES(S,S') = \frac{NED(\text{"string","string"})+NED(\text{"search","research"})}{2} = 0.875$ Similarly,

$AGES(S', S) = \frac{NED(\text{"string","string"}) + NED(\text{"research","search"}) + NED(\text{"matching","string"})}{3} = \frac{2}{3}$
Fuzzy token matching is another example of a powerful hybrid-based method to solve the string similarity problem [WLF11]. This method is discussed further in the next subsection as well as Chapter 4, since it is the adopted method in this work.

2.3 Fuzzy Token Matching

Fuzzy Token Matching is a hybrid-based approach. This method works by first tokenizing the strings, obtaining a list or a set of tokens for each string. Unlike the token-based approach, tokens do not need to be identical. Instead, they are considered similar if their edit similarity is above a chosen threshold δ. The similarity of two strings is then based on the set $T \tilde{\cap}_\delta T'$. This set is obtained by first calculating the similarity of each pair of tokens (t, t') with $t \in T$ and $t' \in T'$. Afterwards, a weighted bipartite graph (bigraph) $G((T, T'), E)$ is constructed. (T, T') are the vertices representing each token. E represents the set of weighted edges between T and T'. Each weight is based on the edit similarity of its two vertices. Next, using the bigraph, a maximum weight matching is found using a matching algorithm like the Hungarian algorithm [Ber81]. In this matching, each token is paired with the closest available token from the other set, therefore obtaining a set of token pairs. Finally, these token pairs are filtered using the given threshold δ to obtain the resulting set or list $T \tilde{\cap}_\delta T'$. The metrics mentioned previously can be adapted for the fuzzy token matching by replacing $T \cap T'$ with $T \tilde{\cap}_\delta T'$.

2.4 Signatures in Fuzzy Token Matching

Maximum weight matching algorithms are expensive. In fact, the Hungarian algorithm has a complexity of $O(n^3)$. Therefore, it is best to avoid unnecessary comparisons when looking for similar token sets among an important number of sets. Signatures are used to filter out as many token set pairs as possible. Signatures work similarly to hash values. They are generated through signature schemes. Signatures are used based on the following principle: Two token sets T and T', their signature sets are $Sig(T)$ and $Sig(T')$ respectively. If T and T' are similar, then $Sig(T) \cap Sig(T') \neq \emptyset$. By reversing this property, we get: if $Sig(T) \cap Sig(T') = \emptyset$, then T and T are not similar, making their comparison unnecessary [Qin+11]. Different similarity metrics require different signature schemes. Additionally, the quality of the signature scheme affects how often signature sets overlap without the token sets being similar. The choice of signature

scheme and discussion of its quality is further discussed in Chapter 4.

2.5 Isabelle Code Structure

Isabelle is a generic and interactive proof assistant. It allows the user to write mathematical definitions and formulas and prove them in a logical calculus.
The Isar proof language is divided into inner syntax and outer syntax. Inner syntax is used to express the terms, while outer syntax represents the Isabelle theory specifications and proof structure. Inner syntax is used to represent Isabelle types and terms of the logic within the outer syntax[Wen22b].
Isabelle code follows a specific tokenization scheme based on the outer syntax of the language. Each keyword, command literal etc. is considered a token. Therefore, a long term written in inner syntax is also considered a single token. Consequently, simply exporting the text of the source code to detect string similarity does not provide accurate results.
Isabelle code is written in .thy files. A collection of related theories and files is called a session. It essentially resembles a project in common IDE environments.[Wen22a]

3 Related Work

3.1 Clone Detection Schemes for Isabelle Code

In this section, we discuss the previous clone detection approaches used for Isabelle.

3.1.1 Token-Based Clone Detection Using Suffix Trees in Isabelle

A token-based clone detection tool has been implemented for Isabelle. This approach uses suffix trees to detect matching code segments. A suffix tree is a data structure that stores substrings in a tree-like structure. It is widely used in pattern matching and string compression algorithms. The developed tool imports selected sessions and concatenates all their tokens to perform a longest common substring search.
As mentioned in the previous section, this token-based approach works using exact string-equality. Therefore, it can reliably detect clones of type 1 and a few clones of type 2 but it is not suited to detect any other type. [Ham22]

3.1.2 Clone Detection on Isabelle Terms based on Abstract Syntax Trees

A different code clone detection scheme has been implemented for Isabelle with a focus on Isabelle inner syntax terms instead of the Isabelle source code. This approach uses abstract syntax trees to detect clones: instead of comparing the source code of inner syntax strings, their parsed abstract syntax tree structures are compared. The goal of this approach was to find clones of type 3. However, mainly clones of type 1 were found. [Ste23]

3.2 String Matching Approaches

In this section, we introduce some algorithms of string matching.

3.2.1 Running Karp-Rabin Greedy String-Tiling Algorithm

Running Karp-Rabin Greedy String-Tiling (RKR-GST) is a comparison algorithm suggested by Michael Wise. In order to compare two strings, we search for unique matches

of substrings from two token strings. Additionally, we calculate hash values for each token and only compare tokens with the same hash value [Wis93]. This comparison algorithm is used in multiple code clone detection tools, such as YAP3 and JPlag. YAP3 is a tool developed to detect text plagiarism and can be used on different programming languages. JPlag is a tool developed to detect plagiarism in Java code. It can also be used to detect clones of C, C++ and Scheme [KL09].

3.2.2 Fast-join: An efficient method for fuzzy token matching based string similarity join

Fast-Join is a method proposed by Jiannan Wang, Guoliang Li and Jianhua Feng. This method is a hybrid-based string matching algorithm. It uses signature schemes as well as bipartite graph matching to solve the string similarity problem. The fuzzy-token similarity metrics mentioned previously in Chapter 2 were proposed and discussed. A few signature schemes are presented and compared. The algorithm presented in Chapter 4 is based on the Fast-join algorithm. [WLF11]

4 Fuzzy Token Matching in Isabelle

Fuzzy token matching consists of comparing each pair of strings or code blocks to decide whether or not they are similar. Each comparison is based on one of the similarity metrics mentioned previously. Therefore, to perform this comparison, a bigraph matching of the two token sets is performed, and the token pairs with a similarity smaller than δ are filtered. If the similarity value of the two code blocks is greater than a chosen threshold Δ, the blocks are considered clones. This process solves the problem of fuzzy clone detection. A direct approach consists of iterating through every pair of code blocks and comparing them. However, the matching of the bigraph is an algorithm that requires a lot of time. Additionally, the number of token sets to compare can be very high, depending on the sessions included. Therefore, three filtering steps are used to create a set of candidates as well as filter out as many unnecessary comparisons as possible. Once candidates have been chosen, the bigraph matching is applied on them as a verification step. Figure 4.1 showcases how the steps of the algorithm.

In this chapter, we discuss each step of the algorithm. We first discuss how code blocks are created. Then, we introduce a new comparison scheme for individual Isabelle tokens. This comparison scheme is used to evaluate the similarity of two code blocks. Then we discuss how the maximum weight matching is performed. Finally, we introduce three optimization steps: a signature scheme for Isabelle tokens, a weight condition and a relaxed version of the matching.

4.1 Token Markup

In this section, we discuss the topic of dividing the Isabelle files into code blocks. This process is performed in two main steps: first, the source code is divided into separate command spans. Each command span contains a command token as well as the corresponding follow-up tokens. Second, these command spans are then divided into two categories: append commands and main commands. Each append command span is aggregated to the nearest main command span before it. Once all append commands have been aggregated, the result is a set of separate code blocks. This

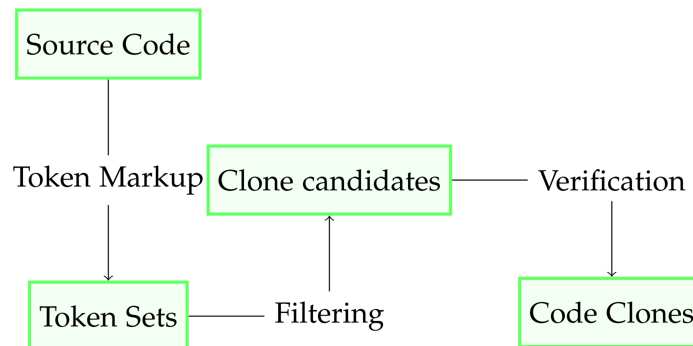

Figure 4.1: Overview of the Fuzzy Token Matching using the Filtering Steps

method is based on the data model in the tool FindFacts [1][HK22]. The resulting data structure has the following parameters:

- **File index** indicates which theory file contains this block.
- **Start and end lines** indicate the line interval of a code block.
- **Token lists** are a representation of the code block. A first list contains all tokens of the command spans. A second list is also created that contains all non-whitespace tokens of the code block. In other words, all tokens that are relevant for the next sections.

As shown in Chapter 2 for token- and hybrid-based string similarity metrics, the size of a string and the similarity of strings are based on the number of tokens. However, inner syntax tokens in Isabelle can be noticeably longer than outer syntax tokens. Therefore, each individual token as well as token list is assigned a `weight` value that represents its practical size. All outer syntax tokens have a weight of 1, whereas inner syntax tokens have a weight equal to their number of constants.

4.2 Token Comparison Scheme

In usual string matching problems, tokens are substrings. However, in Isabelle, a Token is a defined class with two parameters that are relevant to this topic: `kind` is a parameter that indicates the role of the token in the code. `content` is an additional parameter that represents the content of the token itself. Therefore, the token comparator uses these

[1]FindFacts is a search tool for formal Isabelle Theory content. In this work, Isabelle source code is partitioned into text blocks using Isabelle commands: each code block is a set of semantic entities grouped together.

two parameters to determine the similarity of two tokens.

The most common case for token comparison is calculated in two steps: first, the two token kinds are compared to assign a **coefficient** to the comparison at hand. These coefficients indicate how similar the token kinds are. The possible values are 1 for identical kinds, 0.9 for kinds that are very similar and 0 for kinds that are completely different. The idea behind coefficients is to consider the possibility that two tokens with different kinds can still be similar. For example, the different string-type or number-type tokens are similar. If the coefficient for two tokens has been calculated and is 0, the token similarity will return 0 by default. If not, the result of the comparison is the product of the coefficient and the content edit similarity.

In some cases, however, the contents of the two tokens are not important for the token similarity. For example, when variables have names that are completely different, but the code blocks containing them are similar. These cases get assigned a minimum value. Another example would be command tokens that represent similar functionalities such `lemma` and `theorem`. We assigned predetermined values to handle these specific cases.

The tokens with the kind `CARTOUCHE` are inner syntax tokens. Their comparison is different from the rest of the tokens. The method for comparing these tokens falls outside of the scope of this work because their term trees can be compared in a structured way. Therefore, a placeholder method is used to evaluate the similarity of inner syntax strings: they are divided into a set of substrings referred to as constants. The similarity value is calculated using the formula:

$$\text{InnerSyntaxSimilarity}(t_1, t_2) = \frac{|const_1 \cap const_2|}{max(|const_1|, |const_2|)}$$

The token comparison scheme is summarized through Figure 4.2. In this flowchart, the process for comparing token kinds checks whether the kinds are similar. Additionally, the IS comparison process currently represents the placeholder implementation, whereas the normal comparison represents all other non-null cases mentioned above. Once token comparison is possible, the next step is to compare code blocks. Their similarity is evaluated by performing the matching of tokens in their given token lists.

4.3 Token Matching

Two given token sets are compared using a bigraph matching. To perform this, a bigraph is constructed. The vertices are the tokens of the two code blocks. An edge connecting

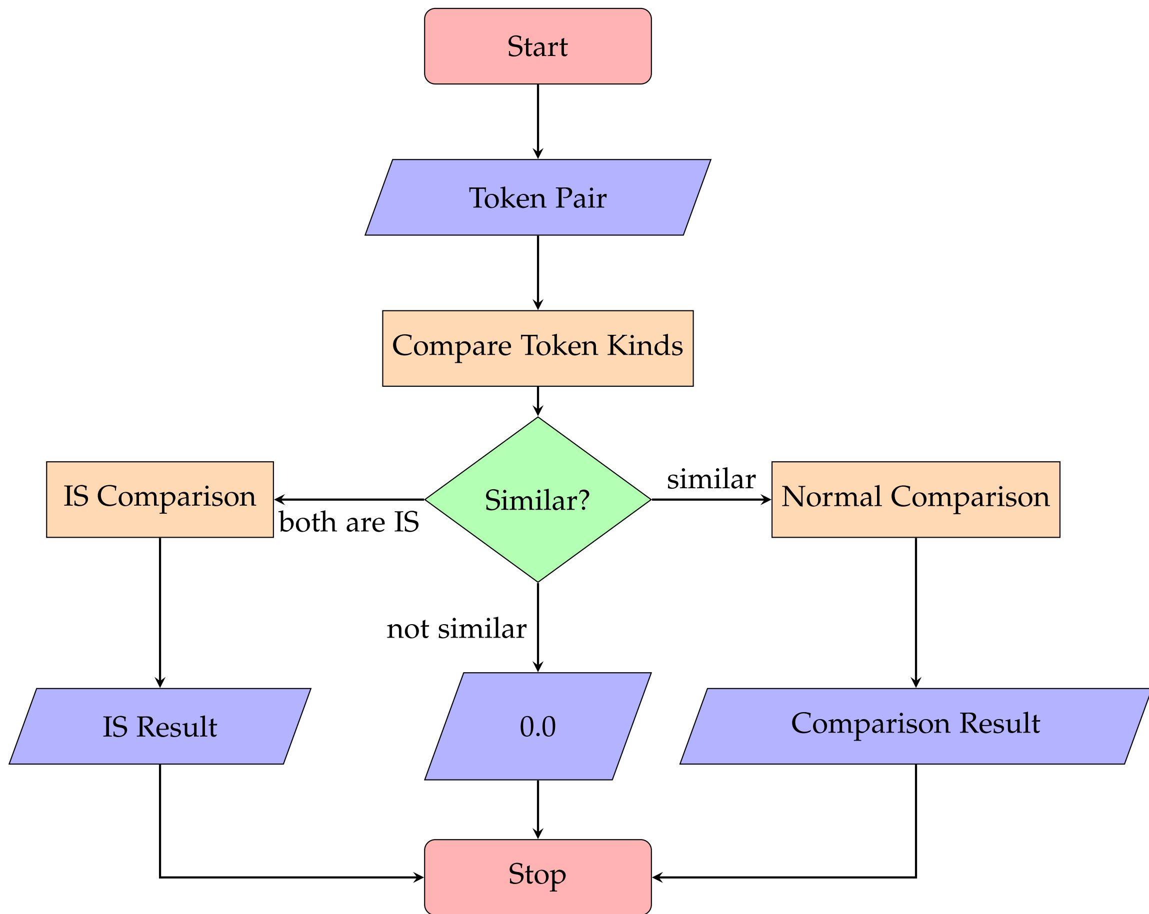

Figure 4.2: Flowchart for the Token Comparison Scheme

two vertices uses the similarity of their tokens as a weight. For the implementation, the graph is represented using its cost matrix. The result of the matching is a set of token pairs, where each token from one set is matched to the closest available token with the highest weight from the other set. The algorithm used to perform the matching is the Hungarian algorithm with a complexity of $O(n^3)$ [Ber81], which is a common algorithm to compute the maximum cost matching. The similarity of the code blocks is evaluated using the fuzzy-Jaccard similarity. If the results are above the chosen threshold Δ, the code blocks are considered clones. When defined in Chapter 2, the fuzzy-Jaccard similarity is based on the cardinality of each token list. In this work, we use the weight of each code block instead:

$$FJACCARD_\delta(S, S') = \frac{weight(T \,\widetilde{\cap}_\delta\, T')}{weight(T) + weight(T') - weight(T \,\widetilde{\cap}_\delta\, T')}$$

4.4 Optimizations

The matchings with a high enough similarity are the final result of the fuzzy token matching. However, matching algorithms including the Hungarian algorithm have a high run time complexity. Additionally, the number of comparisons to perform scales quickly with the number of code blocks to compare. In fact, for n code blocks, the number of comparisons is $\frac{n \cdot (n-1)}{2}$. These factors lead to very long run times. Therefore, the number of comparisons is reduced through the use of a signature scheme, a condition connecting the weights of two code blocks and a relaxed version of the matching algorithm. After these filtering steps are performed, a token matching as explained in Section 4.3 is performed for all the remaining code block pairs to filter out the last candidates and obtain the final results. This final step is referred to as the verification step.

4.4.1 Signature Schemes

In order to reduce the running time, signatures schemes are an efficient tool to avoid unnecessary comparisons. The intuition behind signatures is as follows: from each list of tokens, a multi-set of signatures is generated. The signature multi-sets are then compared to determine which code blocks are not similar and which remain as clone candidates. Using this principle, signatures are used to filter out as many pairs of code blocks as possible. As stated in Chapter 2, if two token sets or lists are similar, then their signature multi-sets intersect. However, two code blocks are only considered clones if their similarity is above the chosen threshold Δ. Therefore, the simple existence of an intersection between the signature multi-sets is not enough to consider a pair of

code blocks to be candidates. Instead, a threshold θ for the size of the intersection is computed to eliminate more code block pairs. In this subsection, the signature scheme for Isabelle code is explained, as well as the choice of two possible signature intersection thresholds.

Signature Scheme for Isabelle Code

The signature scheme works as follows: the tokens of a code block are iterated through, calculating the signature or signature multi-set of each token individually. The signature multi-set of a code block is the union of all signatures of its tokens.
The core concept of an individual token signature is to generalize it without a loss of its meaning. In other words, if two tokens play a similar role, their signature is the same. Whereas two tokens that play different roles get assigned different signatures. This is accomplished using the `Signature_Value` enum. This enum represents some sets of simple and important values. For example, the commands {`lemma`, `theorem` and `corollary`} are represented using the value `LEMMA`, natural numbers and floats are represented using the value `NUMBER` and all type identifiers are represented using the value `IDENT` etc. Additionally, inner syntax constants are all represented using the `CONST` value. The actual token signature is an instance of the `Token_Signature` class. This class has two main parameters: a `Signature_Value` instance as well as an additional string. For a given outer syntax token, its signature is defined using the corresponding enum value and an empty string. For an inner syntax string, each constant has its own signature as well as the string representation of the constant itself. This implies that every inner syntax string gets assigned a multi-set of signatures. Table 4.1 showcases examples for different tokens. As explained previously, the two commands `lemma` and `theory` are assigned the same signature due to a similar functionality. Meanwhile, `section` serves a different purpose and therefore gets assigned a different signature. Additionally, as shown in the last example, the inner syntax token is assigned a signature for each constant.

Intersection Threshold

In order for two code blocks to be considered candidates this inequality needs to be valid:

$$weight(T \mathbin{\widetilde{\cap}_\delta} T') > \Delta \cdot max(weight(T), weight(T'))$$

As explained in the previous paragraph, each outer syntax token gets assigned exactly one signature, whereas an inner syntax string with n constants gets assigned n signatures, with $n \geq 1$. Therefore, $|Sig(T)| \geq |T|$ and $|Sig(T)| = weight(T)$.
We propose the intersection threshold θ_1. It is calculated for two given token lists T_1

Token(COMMAND, "lemma")	Signature(LEMMA)
Token(COMMAND, "theorem")	Signature(LEMMA)
Token(COMMAND, "section")	Signature(SECTION)
Token(IDENT, "x")	Signature(IDENT)
Token(CARTOUCHE,"<A/><rightarrow/><B/>")	Signature(CONST,"A") Signature(CONST,"rightarrow") Signature(CONST,"B")

Table 4.1: Examples of Signatures for Different Tokens

and T_2 as follows:

$$\theta_1 = \Delta \cdot max(|T_1|, |T_2|)$$

We propose a second intersection threshold θ_2. This variant uses the following properties: on the one hand, if two outer syntax tokens t_1 and t_2 match, then $sig(t_1) = sig(t_2)$. On the other hand, if t_1 and t_2 are IS tokens and they match, then:

$$\text{IS_Similarity}(t_1, t_2) \geq \delta$$

This property can be rewritten as follows:

$$|const_1 \cap const_2| \geq \delta \cdot max(|const_1|, |const_2|)$$

Therefore, the intersection threshold θ_2 for two token lists T_1 and T_2 is calculated using the following formula:

$$\theta_2 = \Delta \cdot max(sim(T_1), sim(T_2))$$

with $sim(T)$ being the smallest number of signatures that can lead to a perfect match with T. For a token list T with n_1 outer syntax tokens $\{t_1 ... t_{n_1}\}$ and n_2 IS tokens $\{IS_1 ... IS_{n_2}\}$, $sim(T)$ is calculated using the following formula:

$$sim(T) = \sum_{i=1}^{n_1} weight(t_i) + \sum_{i=1}^{n_2} \delta \cdot weight(IS_i) = n_1 + \sum_{i=1}^{n_2} \delta \cdot weight(IS_i)$$

Furthermore, $\theta_2 \geq \theta_1$, making it a stricter threshold. That means it leads to a smaller number of candidates. However, the formula is slower to compute. The performance of the two variants is compared in Chapter 5.

4.4.2 Weight Condition

In order for two code blocks to be considered similar, the result of the similarity function should be higher than a given threshold $\Delta \in [0, 1]$. This condition can be expressed

using the **fuzzy-Jaccard** similarity metric as follows:

$$\frac{weight(T \mathbin{\widetilde{\cap}_\delta} T')}{weight(T) + weight(T') - weight(T \mathbin{\widetilde{\cap}_\delta} T')} > \Delta$$

The relation between the weights of T and T' is as follows:

$$\Delta \cdot weight(T') < weight(T) < \frac{1}{\Delta} \cdot weight(T')$$

In other words, a token set T' can only be similar to a token set T if the following property is valid:

$$weight(T') \in [\Delta \cdot weight(T), \frac{1}{\Delta} \cdot weight(T)]$$

Therefore, any pair of code blocks that does not fulfill this condition is pruned.

Proof.

$$\frac{weight(T \mathbin{\widetilde{\cap}_\delta} T')}{weight(T) + weight(T') - weight(T \mathbin{\widetilde{\cap}_\delta} T')} > \Delta$$

$$\implies weight(T \mathbin{\widetilde{\cap}_\delta} T') > \Delta \cdot (weight(T) + weight(T') - weight(T \mathbin{\widetilde{\cap}_\delta} T'))$$

$$\implies (1 + \Delta) \cdot weight(T \mathbin{\widetilde{\cap}_\delta} T') > \Delta \cdot weight(T) + \Delta \cdot weight(T')$$

$$\implies \Delta \cdot weight(T) < (1 + \Delta) \cdot weight(T \mathbin{\widetilde{\cap}_\delta} T') - \Delta \cdot weight(T')$$

Additionally, we have the property: $weight(T \mathbin{\widetilde{\cap}_\delta} T') < weight(T')$

$$\implies \Delta \cdot weight(T) < (1 + \Delta) \cdot weight(T') - weight(T')$$

$$\implies weight(T) < \frac{1}{\Delta} \cdot weight(T')$$

□

4.4.3 Relaxed Matching

To reduce the run-time of the matching phase of the algorithm, the matching can be relaxed as follows: each token gets paired up to the closest token of the other set, even if it has already paired up with another token already. Because of this relaxed method, the **Fuzzy-Jaccard** similarity threshold Δ is easier to reach, which potentially leads to false positives. However, if the relaxed matching does not reach Δ, then the original matching cannot reach it either. In other words, this does not introduce false negatives to the results.

4.5 Summary

A summary of the implemented algorithm is represented in Figure 4.3. First, theory files are parsed and divided into code blocks represented by their token lists. Candidate clones are then selected when their signature multi-sets have an intersection greater than θ. These candidates are then further trimmed down using the weight condition as well as relaxed matching. Finally, the remaining candidates are evaluated using the bigraph matching. If their similarity is above the chosen threshold Δ, they are considered clones.

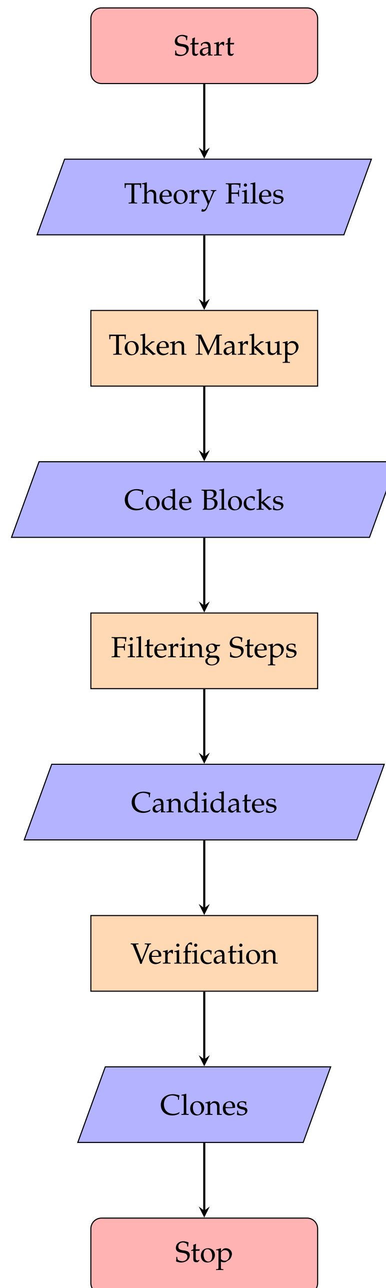

Figure 4.3: Flowchart Summarizing the Optimized Fuzzy Token Matching Algorithm

5 Result Evaluation

We evaluate the results on two levels: first the results are evaluated on a qualitative level. This means how close two code blocks are for different similarity values. The second evaluation is for the speed of the algorithm. This includes discussing the effects of the different filtering steps as well as the speed of the algorithm with and without using the filtering steps.

5.1 Analysis of Sample Results

In this section, the results of the clone detection will be evaluated. Examples of different code blocks with different similarity values are examined. After a short series of qualitative experiments, we observed that token similarity thresholds $\delta = 0.85$ leads to more reliable results. In other words, this threshold allows for the individual tokens to have a fuzzy similarity instead without being very different and without needing to be identical.
The example in Figure 5.1 shows a clone of type 1. This clone consists of the code blocks `conj_cong` in lines $(330, 338)$ and `conj_cong2` in lines $(339, 347)$ in the file IFOL.thy. The fuzzy-Jaccard similarity of these two blocks is equal to 1. That means that all tokens in both code blocks have a match with a similarity higher than $\delta = 0.85$. The code blocks are nearly identical. In fact, there are two small differences: first, the names of the lemmas have a difference of 1 character. Second, the order of the constants `P` and `Q` as well as `P'` and `Q'` is swapped in the fourth line of both code blocks.
A second example of a code clone is shown in Figure 5.2. In this example, the lemmas are `impCE` and `impCE'` in FOL.thy in lines $(83, 92)$ and $(98, 108)$ respectively. The similarity of the code blocks using the Fuzzy-Jaccard similarity measure is 0.96. In this case, the lemmas are clones of type 2. This can be seen through the different names of the lemmas as well as the swapped names of `r1` and `r2`.
The two previous examples show that high similarity values indicate code clones of types 1 and 2
The Figure 5.3 shows the lemma `conj_cong` again along with the lemma `imp_cong` in lines $(355, 363)$ in IFOL.thy. The fuzzy-Jaccard similarity for these blocks is 0.55. These lemmas lead to different outcomes. Therefore they are not clones. However, the two

```
lemma conj_cong:
  assumes ‹P ⟷ P'›
    and ‹P' ⟹ Q ⟷ Q'›
  shows ‹(P ∧ Q) ⟷ (P' ∧ Q')›
  apply (insert assms)
  apply (assumption | rule iffI conjI | erule iffE conjE mp | iff assms)+
  done

lemma conj_cong2:
  assumes ‹P ⟷ P'›
    and ‹P' ⟹ Q ⟷ Q'›
  shows ‹(Q ∧ P) ⟷ (Q' ∧ P')›
  apply (insert assms)
  apply (assumption | rule iffI conjI | erule iffE conjE mp | iff assms)+
  done
```

Figure 5.1: Example of two code blocks in IFOL.thy with a similarity value of 1.0

```
lemma impCE:
  assumes major: ‹P ⟶ Q›
    and r1: ‹¬ P ⟹ R›
    and r2: ‹Q ⟹ R›
  shows ‹R›
  apply (rule excluded_middle [THEN disjE])
   apply (erule r1)
  apply (rule r2)
  apply (erule major [THEN mp])
  done
```

```
lemma impCE':
  assumes major: ‹P ⟶ Q›
    and r1: ‹Q ⟹ R›
    and r2: ‹¬ P ⟹ R›
  shows ‹R›
  apply (rule excluded_middle [THEN disjE])
   apply (erule r2)
  apply (rule r1)
  apply (erule major [THEN mp])
  done
```

Figure 5.2: Example of two code blocks in FOL.thy with a similarity value of 0.96

lemmas have multiple common lines. This indicates that code blocks with a lower similarity contain similar parts. Which means that it is possible to find clones of type 3, if the functionality or result of the blocks is the same.

5.2 Analysis of the Filtering Steps

In this section, the efficiency of the filtering steps is evaluated. This analysis takes two factors into consideration: the number of filtered elements and the speed of the algorithm. After short experiments, we observed that smaller code blocks with a basic structure are often considered clones. Therefore, only code blocks with weights greater or equal to 40 are taken into consideration. Furthermore, the similarity thresholds in these tests are $\delta = 0.85$ and $\Delta = 0.8$

```
lemma conj_cong:
  assumes ‹P ⟷ P'›
    and ‹P' ⟹ Q ⟷ Q'›
  shows ‹(P ∧ Q) ⟷ (P' ∧ Q')›
  apply (insert assms)
  apply (assumption | rule iffI conjI | erule iffE conjE mp | iff assms)+
  done
```

```
lemma imp_cong:
  assumes ‹P ⟷ P'›
    and ‹P' ⟹ Q ⟷ Q'›
  shows ‹(P ⟶ Q) ⟷ (P' ⟶ Q')›
  apply (insert assms)
  apply (assumption | rule iffI impI | erule iffE | erule (1) notE impE | iff assms)+
  done
```

Figure 5.3: Example of two code blocks in IFOL.thy with a similarity value of 0.55

5.2.1 Analysis of the pruned possibilities

In this step, the efficiency of the filters is evaluated from a quantitative standpoint. We analyze how many possible combinations of code blocks are pruned using the different filtering steps. The two variants explained in section 4.4.1 are analyzed and compared. The Table 5.1 and Table 5.2 represent the number of code blocks as well as the number of possibilities initially and after each step of the clone detection algorithm.
As shown in Table 5.1, the use of the threshold θ_1 prunes at least 83% of the possibilities. The weight condition further reduces the number of candidates, whereas the relaxed matching only leaves a few non-clone candidates left.
Table 5.2 shows different values. Since $\theta_2 \geq \theta_1$, the number of of candidates after using signatures is noticeably smaller than in variant 1. In other words, at least 90% of the candidates are filtered. A further consequence of the stricter threshold is the result of the weight condition. In fact, it leads to a negligible number of pruned elements, making it obsolete. Therefore, it is not used in the next tests for this variant.

5.2.2 Influence on the speed of the clone detection

In this subsection, the speed of the implemented algorithm is evaluated. The evaluation is based on performance benchmarks. All tests were performed on a Ubuntu Linux machine with 12GB of RAM. The benchmark tests were done while only using the token matching to detect clones. The remaining tests were done on the same sessions using both variants of the optimized algorithm. Furthermore, seeing as the weight condition is obsolete in variant 2, it was only used to measure the time for variant 1. All results are displayed in Figure 5.4. This bar graph represents the different run-times of the aforementioned algorithm versions for the sessions `Sequents`, `FOL`, `CCL` and `ZF`. The run-

Session	Sequents	FOL	CCL	ZF	HOL
N° of command spans	27	49	96	356	2538
N° of possible combinations	351	1176	4560	63190	3219453
Signature candidates	37	210	397	7165	88587
Weight condition filtering	25	149	227	7013	76987
Relaxed matching filtering	6	3	9	23	148
N° of clones found	6	3	9	18	106

Table 5.1: Number of possible candidates in each step using the signature threshold variant 1 for different sessions

Session	Sequents	FOL	CCL	ZF	HOL
N° of command spans	27	49	96	356	2538
N° of possible combinations	351	1176	4560	63190	3219453
Signature candidates	13	20	73	6766	64529
Weight condition filtering	13	19	72	6765	64507
Relaxed matching filtering	6	3	9	23	148
N° of clones found	6	3	9	18	106

Table 5.2: Number of possible candidates in each step using the signature threshold variant 2 for different sessions

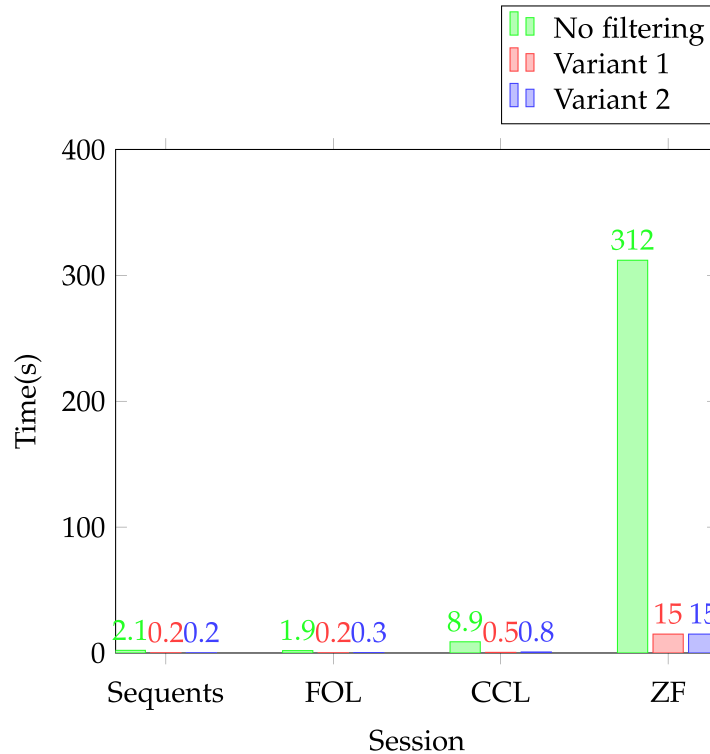

Figure 5.4: Bar graph of the running times of the 3 versions of the algorithm with corresponding optimizations for multiple sessions

times are displayed in seconds on the y-axis. These figure show great improvements in the runtimes of the different sessions. In fact, the optimized versions of the algorithms consistently take less than 10% of the time needed for the non-optimized version. Further tests were performed on the session `HOL`. The run-times of these tests were 716 seconds and 713 seconds for variants 1 and 2 respectively. After running for longer than 2 hours, the test led to an out of memory error for the non-optimized version. These tests indicate that the optimizations improve both memory usage as well as speed of the algorithm. The difference in running time for both optimized variants is negligible in all the sessions, indicating a comparable performance.

6 Conclusion

In this thesis, we implemented a fuzzy token matching algorithm for Isabelle code. We first implemented a token comparator that takes the different token kinds, edit similarity of their contents and further specific cases into consideration. Then we evaluated the similarity of two code blocks by performing a maximum weight matching on their respective token lists. Despite precise results, the matching algorithm was very costly and slow. Therefore, we implemented three main optimization phases, which are filtering steps. First, we implemented a signature scheme for Isabelle. We use signatures to determine whether two code blocks are clone candidates. Second, we implemented a condition connecting the weights of two code blocks with one another. Any pair of code blocks that does not fulfill this condition is pruned. Third, we implemented a relaxed but cheaper version of the token matching algorithm to further reduce the number of candidates. Afterwards, the maximum weight matching is performed on the remaining candidates to verify if they are clones or not. This final step is called the verification step. The results of the clone detection algorithm can be divided into two groups depending on the similarity value of the code blocks. The code blocks with a similarity value greater or equal to 0.85 were copy-paste clones as well as renamed clones. The code blocks with a similarity between 0.5 and 0.6 lead to inconsistent results: they are often functionally unrelated but contain similar parts and lines. The different optimizations of the algorithm led to great improvements in the running time of the algorithm. In fact, the optimizations consistently prune at least 90% of the possible code block combinations and make the clone detection scheme more than 10 times faster. For example, the clone detection on HOL, a session with a lot of code blocks takes longer than 2 hours without optimizations, whereas it takes 12 minutes using the filtering steps.

7 Future Work

This chapter is divided in two parts: further improvements to the proposed algorithm and future features related to the clone detection component.

7.1 Improvements to the Proposed Algorithm

The algorithm developed and implemented is a prototype for fuzzy token matching in Isabelle. The following points can be further improved:

Fine-tuning of the parameters
Throughout the span of this thesis, a short qualitative experiment was conducted and the following parameters were used:

- Minimum code block weight = 40
- Fuzzy-token similarity threshold $\delta = 0.85$
- Fuzzy-Jaccard similarity threshold $\Delta = 0.8$

Further tests and experiments can be conducted to evaluate which values for these parameters can lead to the most reliable and consistent results. Additionally, the precision of the results provided by the fuzzy-Jaccard similarity can be compared with other similar metrics such as the fuzzy-Dice and fuzzy-Cosine similarity.

Improvement of the token comparison scheme
The token comparator relies on textual differences between the different tokens as well as a few specific hard-coded cases. A further variety of cases can be added to include more similar commands and keywords. Additionally, the selected hard-coded cases need to be adapted to accommodate the selected parameters, specifically the token similarity threshold δ. Furthermore, the current inner-syntax comparison scheme is a placeholder method. An appropriate comparison scheme for these specific tokens can be implemented to reliably compare them.

Further optimization of the algorithm
The different filtering steps lead to significant improvements to the speed of the

algorithm. However, it can be further improved through the use of parallel computing. Additionally, the proposed signature scheme in this thesis is a proof of concept. The different signature values can be modified or expanded. This means that more signature values can be added to the current possible ones to represent further possible keywords or commands and lead to a lower number of candidates. Furthermore, two different signature intersection thresholds θ were proposed. Their efficiency can be compared or further threshold values can be proposed. In the presented implementation, we iterate through every combination of code blocks and compare their signatures. This step can be compared with the use of a lookup table.

7.2 Outlook and Further Features

In this section, we introduce a few ideas for the future of clone detection in Isabelle, once the algorithm is refined.

Integration into CI pipelines
Despite the improvements and optimizations, the clone detection algorithm still takes 12 minutes for the session HOL. However, it is unnecessary to run the search on the whole session every time. In other words, using a CI pipeline, older clones can be saved and we only run the clone detection algorithm for code blocks that get modified or added. In that case, the number of comparisons is considerably lower: we only need to perform up to n comparisons instead of $\frac{n \cdot (n-1)}{2}$.

Handling of clones
Once clones are identified, they need to be handled accordingly. For example, if two code blocks have a similarity of 1, only one version of that code block should remain in the session. However, handling code blocks that have a similarity close to 1 is more complex. A possible approach would be notifying the user and letting them decide whether to delete one of the blocks or not.

List of Figures

2.1 Example base pseudo-code 3
2.2 Example code clones of the four main types 3

4.1 Overview of the Fuzzy Token Matching using the Filtering Steps 11
4.2 Flowchart for the Token Comparison Scheme 13
4.3 Flowchart Summarizing the Optimized Fuzzy Token Matching Algorithm 19

5.1 Example of two code blocks in IFOL.thy with a similarity value of 1.0 . 21
5.2 Example of two code blocks in FOL.thy with a similarity value of 0.96 . 21
5.3 Example of two code blocks in IFOL.thy with a similarity value of 0.55 22
5.4 Bar graph of the running times of the 3 versions of the algorithm with corresponding optimizations for multiple sessions 24

List of Tables

4.1 Examples of Signatures for Different Tokens 16

5.1 Number of possible candidates in each step using the signature threshold variant 1 for different sessions 23
5.2 Number of possible candidates in each step using the signature threshold variant 2 for different sessions 23

Bibliography

[Ain+19] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool. "A Systematic Review on Code Clone Detection." In: *IEEE Access* 7 (2019), pp. 86121–86144. ISSN: 21693536. DOI: 10.1109/ACCESS.2019.2918202.

[Ber81] D. P. Bertsekas. "A new algorithm for the assignment problem." In: *Mathematical Programming* 21 (1981), pp. 152–171.

[GF13] W. H. Gomaa and A. A. Fahmy. "A survey of text similarity approaches." In: *International Journal of Computer Applications* 68 (2013).

[Ham22] M. Hamacher. "Development and Integration of a Clone Detection Tool for Isabelle/Isar." In: (2022).

[HK22] F. Huch and A. Krauss. *FindFacts: A Scalable Theorem Search*. 2022. arXiv: 2204.14191 [cs.LO].

[KL09] C. Kustanto and I. Liem. "Automatic source code plagiarism detection." In: *10th ACIS Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD 2009, In conjunction with IWEA 2009 and WEACR 2009* (2009), pp. 481–486. DOI: 10.1109/SNPD.2009.62.

[Qin+11] J. Qin, W. Wang, Y. Lu, C. Xiao, and X. Lin. "Efficient exact edit similarity query processing with the asymmetric signature scheme." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2011), pp. 1033–1044. ISSN: 07308078. DOI: 10.1145/1989323.1989431.

[RBS13] D. Rattan, R. Bhatia, and M. Singh. "Software clone detection: A systematic review." In: *Information and Software Technology* 55.7 (2013), pp. 1165–1199. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2013.01.008.

[RC07] C. K. Roy and J. R. Cordy. "A Survey on Software Clone Detection Research *." In: (2007).

[Ste23] A. Steinhauer. "Clone Detection on Isabelle Terms based on Abstract Syntax Trees." In: (2023).

[Wen22a] M. Wenzel. "The Isabelle System Manual." In: (2022).

[Wen22b] M. Wenzel. "The Isabelle/Isar Reference Manual." In: (2022).

[Wis93] M. J. Wise. "Running Karp-Rabin Matching and Greedy String Tiling Neweyes: A System for Comparing Biological Sequences Using the Running Karp-Rabin Greedy String-Tiling Algorithm 1 Neweyes: A System for Comparing Biological Sequences Using the Running Karp-Rabin Greedy String-Tiling Algorithm." In: (1993).

[WLF11] J. Wang, G. Li, and J. Fe. "Fast-join: An efficient method for fuzzy token matching based string similarity join." In: *Proceedings - International Conference on Data Engineering* (2011), pp. 458–469. ISSN: 10844627. DOI: 10.1109/ICDE.2011.5767865.

bibliography